

**UNCLASSIFIED**

**Copy No:** \_\_\_\_

**C<sup>4</sup>I Branch**

**Guidebook for Software Metrics**

**October 1995**

**UNCLASSIFIED**

## CONTENTS

<u>Section</u>	<u>Page</u>
LIST OF TABLES	III
LIST OF FIGURES	III
1. INTRODUCTION	1
1.1 Scope	3
1.2 Selection Criteria	4
1.3 Organization	4
1.4 Applicability	4
1.5 Reporting	7
1.6 Acknowledgments	7
2. BACKGROUND	8
2.1 Why Use Software Measurement	8
2.2 Software Measurement Process Concepts	8
3. SOFTWARE MEASURES	12
3.1 Software Size	15
3.2 Software Complexity	23
3.3 Software Schedule	34
3.4 Software Development	38
3.5 Software Testing	43
3.6 Software Builds	47
3.7 Software Defect Reporting	50
3.8 Software Effort	57
3.9 Software Problem Resolution Effort	61
4. REFERENCES	66
APPENDIX A : ACRONYMS	70
APPENDIX B : TERMS	72
APPENDIX C : SOFTWARE DEFECT REPORT GUIDELINES	74
APPENDIX D : OBJECT-ORIENTED METRICS	81
INDEX	83

<u>List Of Tables</u>	<u>Page</u>
Table 1-1 The Metrics	2
Table D-1 Object Oriented Metrics	71

<u>List Of Figures</u>	<u>Page</u>
Figure 1.1-1 Metrics/Phase Coverage	6
Figure 3-1 Metrics Template And Field Descriptions	13
Figure 3-2 Generic Life-Cycle Metrics Graphical Report Form	14
Figure 3.1.2-1 Software Size	18
Figure 3.1.2-2 Function Point Monthly Report	21
Figure 3.2.2-1 Average Software Unit Complexity	26
Figure 3.2.2-2 Average CSCI Complexity	27
Figure 3.3.2-1 Software Schedule	36
Figure 3.4.2-1 Software Development Progress	40
Figure 3.4.2-2 Software Development Progress (Object-Oriented)	41
Figure 3.5.2-1 Software Test Progress	45
Figure 3.6.2-1 Software Build Progress	49
Figure 3.6.2-2 Software Class/Build Progress	52
Figure 3.7.2-1 Software Defect Reports	55
Figure 3.8.2-1 Cumulative Effort	60
Figure 3.9.2-1 Cumulative Average SPR Resolution Effort	63
Figure 3.9.2-2 Closed SPRs	64
Figure 3.9.2-3 Valid SPRs By Phase Found and Category	65

## 1. INTRODUCTION

---

The software engineering metrics presented in this document are intended for Contracting Office Representative (CORs) and managers of acquisition, development, and support projects within NRL's C<sup>4</sup>I Branch. The metrics have been chosen to provide insight into common project issues/questions concerning how well program and project objectives are being met. The metrics, summarized in table 1-1 below, can be organized into three areas based on the objectives they address.

**Resources:** This area includes metrics that address issues/questions about cost control, schedule development, personnel allocations, and accuracy of estimation models. Examples of questions addressed within this objective are:

- How much do size estimates change over a period of time during the development?
- How does the trend of size estimates effect resource allocations?
- What does actual cost and schedule data imply about the validity of underlying estimation models?
- How does productivity related to reused software compare with productivity of newly developed or modified software?
- How is productivity related to the specific implementation language?

Examples of strategic objectives within this objective area include control software development and maintenance costs; improve accuracy of cost, effort, and sizing estimation techniques; efficient allocation of personnel resources; and understand how effort is being spent in each stage of the development or post-deployment software support (PDSS); and understand how computer resources (e.g., CPU timing, memory) are being expended.

**Quality:** This area includes metrics that address issues/questions about the quality of products being produced and the quality of the processes used to produce the products. Examples of questions addressed within this objective are:

- Are test plans/procedures adequate to exercise program segments?
- Are expected problem report densities reflective of recorded program complexities?
- Are modules sufficiently cohesive to promote reuse and to minimize side effects?
- Does the complexity level of the design/implementation support maintainability goals?

Examples of strategic objectives within this area include the ability to increase customer satisfaction; to reduce defects; to reduce rework; to develop maintainable systems; and to improve development and PDSS processes.

**Progress:** This area includes metrics that address issues/questions about the development and PDSS status of a program. Examples of questions addressed within this objective area include:

- Is the contractor meeting commitments on time and within the prescribed level of quality?
- What is the true (objective) progress of the effort?
- Are schedule and forecasts to completion realistic?

- Are functional allocations shifting from earlier to later builds?
- Are planned productivity rates being achieved?
- Are levels of rework having an impact on progress?
- Is requirements instability having an impact on progress?

Examples of strategic objectives within this area include the ability to increase timely delivery of systems; to increase stability of requirements specifications; to improve interface specification processes; to improve Software Unit-level development and PDSS processes; to improve software CSCI/system-level test processes; and to improve the software build processes.

**Table 1-1. The Metrics**

<u>Objective Area</u>	<u>Strategic Objective</u>	<u>Metric</u>	<u>Purpose</u>
Resources	Reduce software development and maintenance costs, improve accuracy of cost / effort / sizing estimation techniques, efficient allocation of resources	Size	Track changes in the magnitude of the software development effort
			Update/tune software development estimation models
	Reduce software development and maintenance costs, improve accuracy of cost / effort / sizing estimation techniques, efficient allocation of resources, understand how effort is being spent in each stage of development	Effort	Monitor total number of staff-months / staff-hours expended on the project
Quality	Develop maintainable systems	Complexity	Update/tune software development estimation models
	Maximize customer satisfaction, minimize defects, improve development/support processes, minimize rework	Software Defects	Track development organization's ability to maintain an acceptable level of complexity at the Unit, and CSCI level
			Track development organization's ability to test a system based on program requirements

**Table 1-1. The Metrics (Continued)**

<u>Objective Area</u>	<u>Strategic Objective</u>	<u>Metric</u>	<u>Purpose</u>
	Maximize customer satisfaction, improve development/support processes, minimize rework	Problem Resolution Effort	Monitor cumulative effort required to resolve problem reports (rework)
Progress	Maximize timely delivery of systems, improve development /support processes	Schedule	Track development organization's ability to maintain the development schedule by tracking the delivery of planned software packages
	Maximize timely delivery of systems, improve Unit-level development and/support processes	Development	Measure the development organization's ability to keep Unit design, coding, test, and integration activities on schedule
	Maximize timely delivery of systems, improve CSCI/System-level test processes	Testing	Measure the development organization's ability to maintain testing progress  Measure efficiency and effectiveness of test program
	Maximize timely delivery of systems, improve software build processes	Software Builds	Monitor the development organization's ability to test a system based on program requirements

## **1.1 Scope**

Currently, the set of metrics address a limited number of issues and strategic objectives. There is an emphasis on progress metrics that highlight deviations between monthly planned and actual values as well as an emphasis on metric presentations that highlight trends. Such metrics and presentations should help stimulate COR-contractor discussions, leading to early identification and resolution of potential management problems. Many of the metrics also reflect work by the MITRE Corporation [SCHULTZ 88], recommendations by the Software Engineering Institute (SEI) [PAULK 91, ROZUM 92], and recent experiences in the use of such measurements by industry/academic [GRADY 94], [LORENZ 93], [LORENZ 94].

The metrics are not limited to specific parts of the software life cycle. They can be applied during initial software development as well as post deployment software support (PDSS). While

the terminology used in this policy is characteristic of a classical DoD waterfall development approach, the metrics are not limited to such an approach, but, rather, are applicable to different development paradigms (e.g., iterative, prototyping, risk-oriented, spiral). Figure 1.1-1 is a summary of when the different metrics are applicable in terms of classical MIL-STD initial development activities -- System Requirements Review (SRR), System Design Review (SDR), Software Requirements Review (SRR), Preliminary Design Review (PDR), Critical Design Review (CDR), Test Readiness Review (TRR), and Physical Configuration Audit (PCA). In Figure 1-1.1, testing activities are assumed to be quite varied. There can be Software Unit testing to verify unit correctness, Computer Software Configuration Item (CSCI) integration to verify interfaces and interaction among the CSCIs, and system-level testing to verify software/hardware interfaces. Complete descriptions of the initial project life-cycle activities and milestones used here can be found in [MIL-STD-498].

## **1.2 Selection Criteria**

Formalization through the SEI of metrics collection and analysis as a key component of a mature software development environment has led to a significant rise in the number of potentially useful measures published in the literature. Although many show promise, the empirical data to demonstrate their value within a product development is still sparse. Consequently, many of these measures are not included in the report but were included in Appendix D for general reference. Filtering of this report was based on the following criteria:

- Must be straightforward to collect
- Must provide a foundation for future definition/evolution of metrics
- Analysis must be susceptible to automation support
- Terminology/metrics must be development model independent
- Value must be demonstrated by practice

## **1.3 Organization**

There are three remaining chapters. Chapter 2 contains background material for managers to effectively use the metrics defined in Chapter 3. Chapter 4 is a list of the references used to generate this report and is a source for additional readings. The four appendixes contain a list of acronyms, definitions of terms used in the document, guidelines for the construction of software defect reporting forms and metrics of potential interest to uses of this report but not meeting the selection criteria established in Section 1.2.

## **1.4 Applicability**

The metrics described in Section 3.0 are generally applicable to all projects. Within a metric category, however, selection of particular measurements may be dependent on such program characteristics as development approach (functional vs. object-oriented), estimation model assumptions (SLOC vs. FP), or application (expert system, data base, etc.). At the beginning of each project, program managers should specify metrics and graphs to be required of the developer. Although tailoring the application of the metrics may be reasonable, it is expected that the core metrics of Size, Effort, Defects, and Schedule be retained. Examples of potentially useful tailoring are:

- replacement of schedule progress metrics with similar metrics collected on predecessor projects,
- replacement, deletion, or consolidation of development milestones,

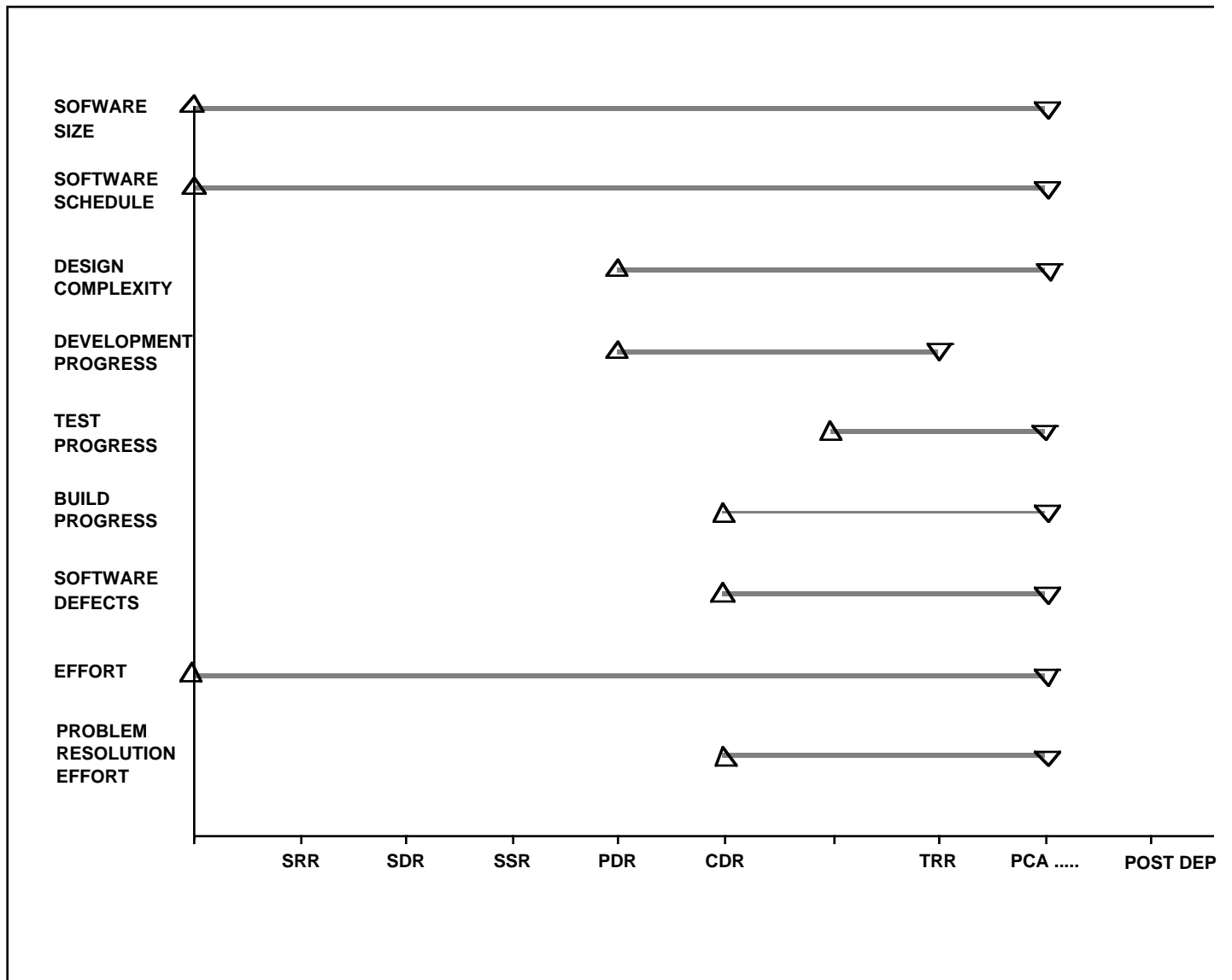


Figure 1.1-1 Metrics/Phase Coverage

- redefinition of terms (e.g., staff-month, source line of code),
- adjustment of triggering or upper control-limit guidelines,
- level of application (e.g., CSCI vice Unit-level, tracking cumulative problem resolution effort for “critical” problems only, monitoring test progress data for critical CSCIs)

## **1.5 Reporting**

Requests For Proposals (RFPs) and Laboratory tasking should specify the metrics to be collected and reported upon, the reporting period/frequency, and the delivery formats (toolsets and platforms). Some examples of statement of work materials are:

“The developer shall provide software metric plans in accordance NRL C<sup>4</sup>I Guidelines for Software Measurement, dated October 1995.

“The set of metrics shall comprise:

- Size
- Effort
- Complexity
- Software Defects
- Problem Resolution Effort
- Schedule
- Development
- Testing
- Software Builds

“Specific reporting/graph formats, triggering levels, and other tailorable metrics data shall be determined during program planning stages, documented in program planning documentation, and presented for review/approval at initial program planning reviews.”

“The developer shall describe the processes used to collect and validate the metrics data in program planning documentation or in the Software Development Plan.”

“The developer shall present updated metrics and graphs at monthly business or technical meeting as appropriate. Cumulative metrics and graphs shall be delivered to the Government at least one (1) week prior to the meetings.”

“Metrics and graphs shall be maintained, produced, and delivered using Microsoft Excel on a Macintosh.”

## **1.6 Acknowledgments**

The principal authors of this report have been James Hager (ARL), Louis Chmura (NRL), Richard Chen (ARL) and Roland Isnor (ARL).

## 2. BACKGROUND

---

This chapter provides background information for C<sup>4</sup>I s/w development managers to understand and use the metrics defined in Chapter 3.

### 2.1 Why Use Software Measurement

A software measurement program provides project managers with a systematic method of measuring, assessing, and adjusting a project's software approach using objective, quantitative data. Without a software measurement program, it can be difficult to:

- understand the current status of a program along its many, related dimensions: cost, schedule, personnel, requirements, quality, maintainability, etc.
- identify and resolve problems in a timely manner

Organizations with successful measurement programs report the following benefits [GRADY 87], [GRADY 94]:

- Better insight into product development
- Capability to quantify tradeoff decisions
- Better understanding of new technology impacts
- Better planning, control, and monitoring of projects
- Better understanding of both the software development process and the development environment
- Better understanding of performance on past projects
- Identification of areas of potential process improvement as well as an objective measure of improvement efforts
- Improved support for maintenance goals
- Improved Government/contractor communication

Additional benefits include improved quality and more timely delivery of software products.

### 2.2 Software Measurement Process Concepts

Rozum [ROZUM 92] identifies five key components to an effective measurement program:

1. Defining clearly the software development issues and questions and the software measures that support insight into the issues/questions.
2. Collecting, validating, and understanding the measurement data.
3. Converting the software measurement data into graphs and tabular reports that support analysis of the issues.
4. Analyzing/correlating the measurement data to provide insight into the program issues.
5. Using the results to support program decision making, to identify new issues and questions, and to ultimately implement process improvements [HUMPHREY 89].

Basili and Weiss [BASILI 84] recommend establishing measurement goals/objectives prior to identifying issues/questions. Example objectives might be, “Characterize project software defects,” and “Evaluate the success in constructing easy-to-change software.” Once the measurement objectives are established, they can be used to develop issues/questions to be answered by the measurement effort. Example questions might be: “How quickly do we resolve defects on the average?”, “Are we getting better at defect resolution?”, “Do changes tend to be confined to single components?”, and “What is the average effort in making a change?” A software measurement program that carefully identifies objectives and relevant issues/questions, and then collects measurement data based on these, should prevent unneeded and excessive data collection while still providing information needed by managers and technical staff.

Once the issues/questions are identified, then required metric data and data categories must be defined. For example, for the case of “effort for making changes,” it is important to define what kinds of personnel hours will be accumulated. If there is a desire to differentiate between simple, modest, and hard changes in terms of effort, then such categories need precise definition. For certain metrics, definition help is available. For example, Goethert et al. [GOETHERT 92] provides guidance for establishing a definition of effort; Park [PARK 92] provides guidance in the area of source line of code (SLOC) counts.

Collecting measurement data often requires that a data collection form be developed and used. Without the use of forms, it would be necessary to rely on people’s memories and constant re-examination of project deliverables and engineering notebooks. Ideally, data collection forms should be kept brief and should be given a trial run before extensive use. Once established, however, project management should require everyone use the forms. And, because metric data collection is often a people process, contractor management must make periodic checks to ensure correct, consistent, and complete measurement. For example, Unit Development Folders [MIL-STD-498], should be sampled periodically to ensure that personnel are counting source lines of code (SLOC) consistently with definitions recorded in the contractor’s software development plan.

In general, metric data needs to be presented as time-based plots [GRADY 87, CHMURA 90]. The graphs should include support for reporting of “raw” measurement data, annotations highlighting discrepancies between actual and expected data, and multiple development/support stages. Organizational techniques that can help the program manager understand and analyze the measurement graphs include [SCHULTZ 88]:

- Partitioning the data to highlight particular differences (e.g., graph by organization, language, CSCI).
- Adding data sources on measurement graphs and reports. Knowing the source and date of the data will help program managers interpret it.
- Adding program major milestones on the graphs to support context analysis. To better understand how the data being reported impacts or relates with the overall development process, major milestones and stages should be consistently annotated on the time axis (SRR, PDR, CDR, TRR, initial development, post deployment software support, etc.).
- Reviewing changes in plans or methods to determine applicable comparisons. Graphs of progress-oriented metrics should include the original baseline along with the currently approved plan and actual data.

Typically, analysis of measurement data must involve all levels of the program management/technical staff in an integrated effort to improve project performance. Measurement data needs to be provided by the contractor, then processed and analyzed by both the contractor

and the government. The contractor should be able to use the data to manage its own processes, to adjust development schedules, and to provide meaningful status reports to the government. Experience has shown that the government must have access to the basic data so it can independently analyze issues, interpret the results, and apply lessons learned to other acquisitions.

Rozum [ROZUM 92] offers some simple guidelines when using collected metrics.

- Use the measures together with other management information to improve insight into progress and risk assessment.
- Use the software measurement program only as the starting point for obtaining insight into program developments. To isolate specific causes of trends indicated by the measures, it is often necessary to ask additional questions. An important byproduct of an effective measurement program is the ability to ask the “right” questions.
- Identify and implement changes to improve the program, including any needed improvements to the measurement process.

There are other common guidelines. One important guideline is not to use data to assign blame, but rather to help identify and resolve potential project problems. A second is not to become preoccupied with the metrics themselves and fail to focus on the project performance issues. For example, if design reviews indicate that a significant number of Software Units are exceeding program specified complexity levels, a manager should not focus on the procedural logic of the affected Software Units. Rather, the concern should be with a contractor’s internal design review process that does not filter high-complexity Software Units. A third guideline is to realize that engineering processes and products are dynamic and subject to change. Requirements become better defined: cost, performance, and schedule estimates are refined; additional personnel, hardware, test, and support resource requirements are identified; and insights from the measures are gained. This will result in some metrics showing significant differences between plans and actuals. Such differences may merely show that a contractor is responsive to evolving program needs (of course, too much change could indicate that the process is not stable or that the requirements are not well understood). A fourth guideline is to expect that, as project measurement data is better understood, it will become apparent that not all data collected is useful because the items are outdated, there is low confidence levels in the validity of the data, or the data are inconsistent with other data items that are considered to be accurate or valid. As the issues evolve, the program manager may need to adapt the measurement process (generally not the fundamental measurement definitions) to address the changes.

THIS PAGE INTENTIONALLY LEFT BLANK

### 3. SOFTWARE MEASURES

---

This chapter defines a set of nine metrics for project managers to measure, assess, and adjust a project's software approach using objective data. Collecting, analyzing, and correlating the recommended data will help project managers understand the following aspects of a program development:

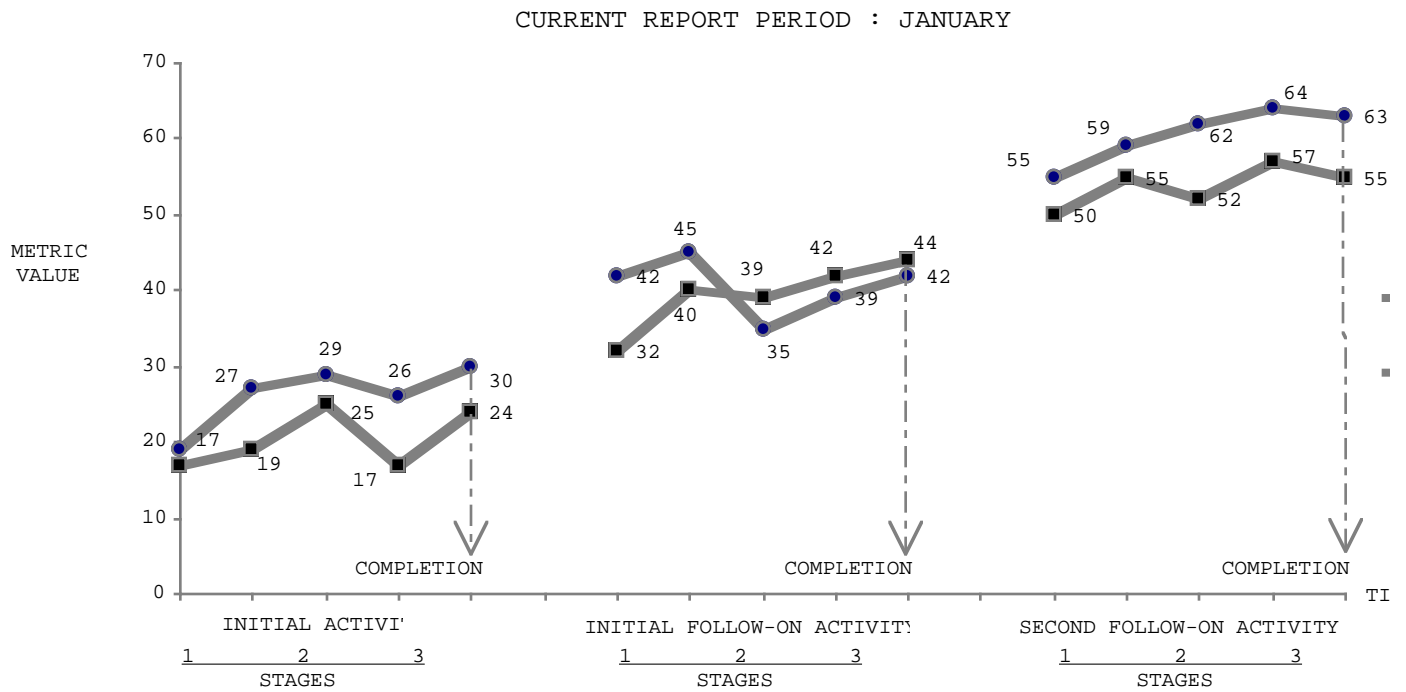
- requirements stability
- staff allocation/stability
- development progress
- adequacy of test program
- product quality
- product maintainability
- level of rework
- adequacy of internal review process
- project effort/spending profile
- project productivity assumptions
- accuracy of size estimation processes
- accuracy of cost/schedule models

The metrics are described using a standard template, that includes the following sections: *Category, Purpose, Input Data, Tracking Period, Frequency of Collection, Usage, Triggering Guideline, Action, Special Definitions, and Notes*. Figure 3-1 provides an example of the metric's templates and a description of the content fields. The Category field corresponds with the objective areas described in Chapter 1. Triggering guidelines identify metric limits, which if exceeded, warrant the attention of project managers. Each template-based description is followed by guidelines on effective use of the metric and by one or more sample time-based graphical presentations of the measurement data.

The metrics can be collected and graphed over the periods of coverage described in Figure 1.1-1. Although the milestones identified in Figure 1.1-1 are reflective of MIL-STD-498 development standards, sampling periods and associated descriptions are activity-specific should be tailored to support project reporting goals. Figure 3-2 provides the generic graph format selected for reporting measurement data. The format for the graphs is flexible and includes support for reporting of "raw" measurement data, annotations highlighting discrepancies between actual and expected data, and multiple development/support activity periods. For example, measurement data collected in the development stages should be merged with post deployment software support (PDSS) data to provide a complete life-cycle profile. The frequency of monitoring is dependent on the size and duration of the project, the life-cycle model selected, and the specific program activities, and should be determined at project startup. For large projects or project activities spanning multiple years, monthly reports are appropriate. For projects of shorter duration, weekly or biweekly reports are more appropriate.

<b>Category:</b>	Objective Area (Resources, Quality, Progress)
<b>Purpose:</b>	Target goals for collecting measurement data
<b>Input Data:</b>	Measurements to be collected each reporting period
<b>Tracking Period:</b>	Applicable life-cycle reporting period(s) (e.g., Analysis, Design, Testing, Post Deployment Support Stage, Full life-cycle)
<b>Frequency of Collection:</b>	Frequency of data collection / reporting during the tracking period (e.g., weekly, monthly, milestone-oriented)
<b>Usage:</b>	General guidelines for interpretation of measurement data
<b>Triggering Guidelines:</b>	Metric thresholds which if exceeded require project management action
<b>Actions:</b>	Specific actions to be taken when measurement data exceeds project specific thresholds
<b>Special Definitions:</b>	Definition of specialized terms used during metric discussions
<b>Notes:</b>	General information useful for understanding metric discussions
<b>Additional Questions:</b>	Potential project management questions to further understand the implications of measurement data on project development goals

Figure 3-1 Metrics Template And Field Descriptions



REPORT PERIOD

STAGES  
1    2    3

CURRENT  
REPORT  
PERIOD DATA

INITIAL ACTIVITY  
INITIAL FOLLOW-ON ACTIVITY  
SECOND FOLLOW-ON ACTIVITY

REPORT STATUS

TRIGGERING REPORT BASED  
ON PROGRAM SPECIFIED VALUES

Figure 3-2 Generic Life-Cycle Metrics Graphical Report Form

## 3.1 Software Size

### 3.1.1 Purpose

Software Size measurements provide CORs and other personnel with an indication of the size and characteristics of the software being developed or modified. Software size indicates the amount of work to be done and the number of resources needed to do the work. Initially software size measurements are estimated in a contractor's proposal. Subsequently, size estimates are updated monthly or during major program reviews to compare actuals with the estimated values. Post delivery size measurements are included in estimation model databases and updated during maintenance stages.

### 3.1.2 Description

METRIC CATEGORY: Resources - Size

PURPOSE: Track changes in the magnitude of the software development effort; provide historical data for software cost estimation models.

INPUT DATA: Estimated New/Reused/Modified Source Lines of Code (SLOC);  
Estimated Total SLOC (sum of categories);  
Actual Total SLOC (sum of categories)  
Function Points (FP)

TRACKING PERIOD: Full Life-cycle

FREQUENCY OF COLLECTION: Monthly/Major Milestone Reviews

USAGE: Sizing measurements recommended in this report are SLOC and FP. Traditionally, model-based estimation processes have relied on SLOC measurements to generate software costing and schedule data [VERNER 92]. More recently, FP analysis has surfaced as a language and application independent estimation approach [SYMONS 91], [MATSON 94], [HUMPHERY 95], and [JONES 94]. In many cases, it is appropriate to combine both approaches to more realistically generate cost and schedule data.

***Source Line Of Code:*** SLOC estimates and actual SLOC data, together with the assumptions from which the estimates were generated, can provide the historical data necessary for improving the cost and schedule estimation process. Examples of templates used to record SLOC assumptions are found in [PARK 92]. Sizing templates provide:

- A checklist form that enables project managers to identify the issues and choices they must address to avoid ambiguity and to communicate precisely what is included and excluded in the size measurements.
- Examples of how to use the checklist to construct specifications that will meet differing objectives.
- Examples of forms for recording and reporting measurement results.

Increase in total SLOC or significant migration among SLOC types (New, Reused, Modified) can lead to schedule slips and staffing problems. Changes in SLOC estimates often result from a better understanding of requirements. However, significant changes must be accounted for in schedule and staffing estimates. Visibility within new, reused and modified code categories must be demonstrated since the potential exists for total code growth counts

to remain constant despite significant fluctuations within subordinate code categories. For example, a 20% growth in new code estimations might be offset by a 20% reduction in modified code counts, resulting in no total code growth. The problem with this loss of visibility is the nature of the code types. The amount of effort required to generate new code is usually greater than the amount of effort to modify existing code. An even greater discrepancy exists between reused code and these categories. For this reason, it is important to track on each category, using project-defined triggering levels to highlight the need to management attention.

Many cost and schedule estimation-model parameters (e.g., productivity, experience, tool support) are language dependent. SLOC measurements can be partitioned by development language (Ada, C, C++, etc.) or aggregated by configuration item (CSCI, Software Unit, etc.) to provide more visibility into potential program milestone or resource problems.

There are no universally correct triggering levels for migration within code categories and estimated total code counts. Establishment of these levels is project specific and may depend on such diverse project characteristics as project size, life-cycle phase, or target application. The 5/10/10% triggering guidelines provide a starting point to investigate the appropriate values. For more critical CSCIs or Software Units it may be necessary to establish more stringent values and track on these elements separately. In some cases, it may be necessary to track code counts for each application language used.

Figure 3.1.2-1 illustrates changes in new code counts that produce total code counts greater than the triggering level. It is important to note that monthly code counts may fall within project-specified windows but small, monthly growth may ultimately trigger cumulative counts. As project efforts transition from the specification and design phases to implementation and testing phases, estimated sizing data becomes more concrete. Fluctuations in code counts experienced during the design phases, stabilize during the unit test, software integration and test, and system integration and testing phases.

**Function Points (FP):** Another method of estimating software size is function points analysis. FPs are calculated by counting the number of system externals and internal files. The five most relevant items that are often counted or estimated in FP calculations are external inputs, outputs, inquiries, interfaces and internal files [HUMPHERY 95].

External inputs are data and/or control inputs entering the external boundary of the system, causing system processing to take place. Some examples of external inputs are input files, input tables, input forms, input screens and input transactions. Input files can include data files and control files while input forms include documents and data entry sheets. Input screens include data screens and functional screens. Input transactions contain control data, interrupts, system messages, and error messages.

External outputs are data and/or control outputs that leave the external boundary of the system after processing has occurred. Examples of this group include output files which are often data files and control files, output reports, and output tables. Output reports also include system messages, error messages and printed and screen reports from a single interrupt.

External inquiries are I/O queries which require an immediate response. Some examples of external inquiries are interrupts, system calls, and prompts.

External interfaces are files or programs which are passed across the external boundary of the system. Specific examples include program libraries (e.g., run-time libraries, package or generic libraries), math libraries (e.g., library of matrix manipulation routines, library of

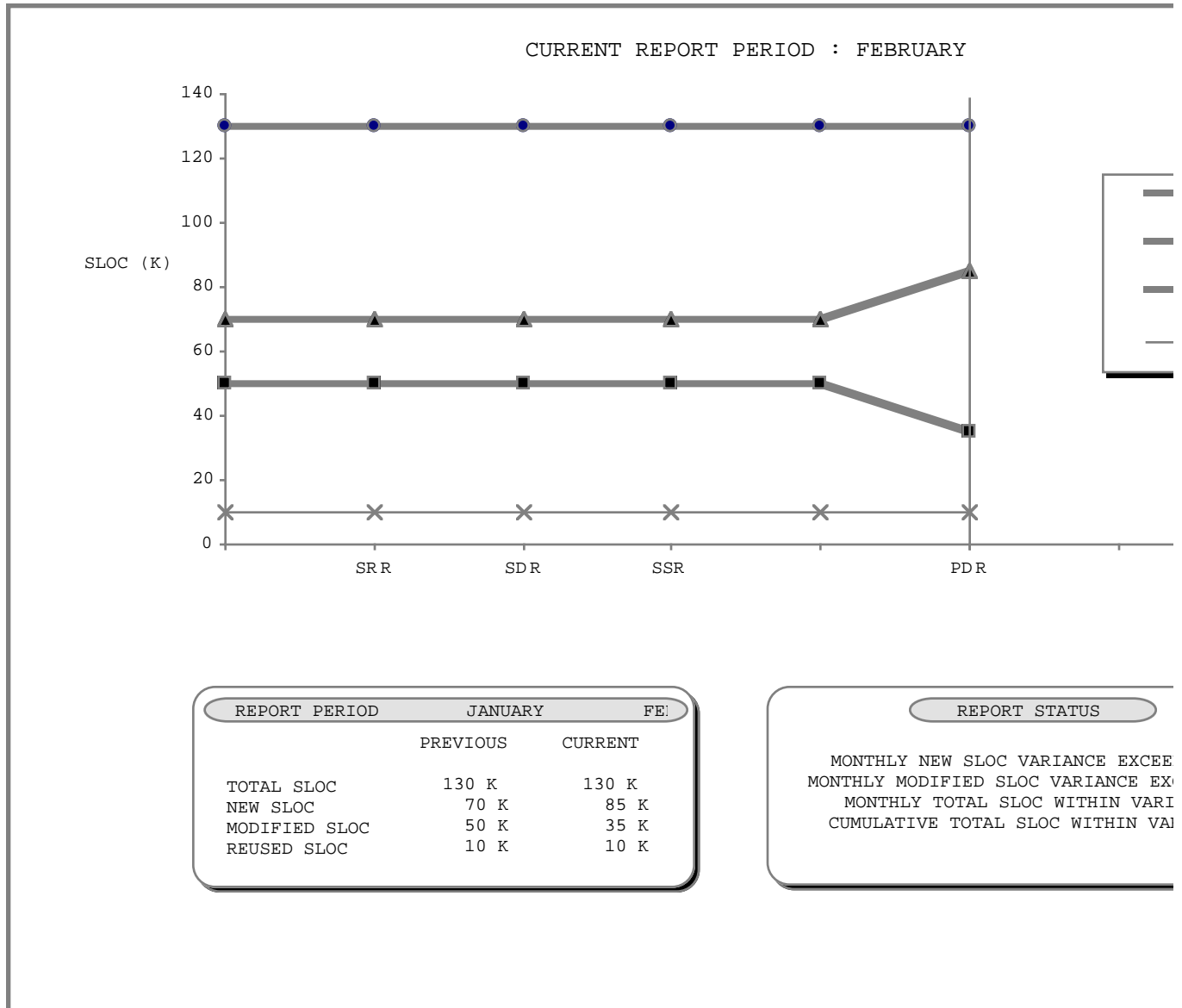


Figure 3.1.2-1 Software Size

coordinate conversion routines), common utilities (e.g., I/O routines, sorting algorithms), shared databases, and shared files.

Internal files are logical grouping of data or control information stored internal to the system. Some examples of internal files are data files, control files, directories, and databases.

Based on these five areas and the particular method of function point calculation, each count is multiplied by an appropriate weight and summed to determine the FP count. Examples of FP counts and their significance can be seen in [JONES 95] and [YEH 93].

Figure 3.1.2-2 provides an example of a FP monthly report. It shows graphical and tabular counts (initial, previous, and current) for each of the FP categories listed above.

**TRIGGERING GUIDELINE:**     5% total SLOC from previous month estimates  
                                         10% migration within SLOC categories  
                                         10% cumulative total SLOC  
                                         10% FP growth within a category  
                                         20% Cumulative FP growth

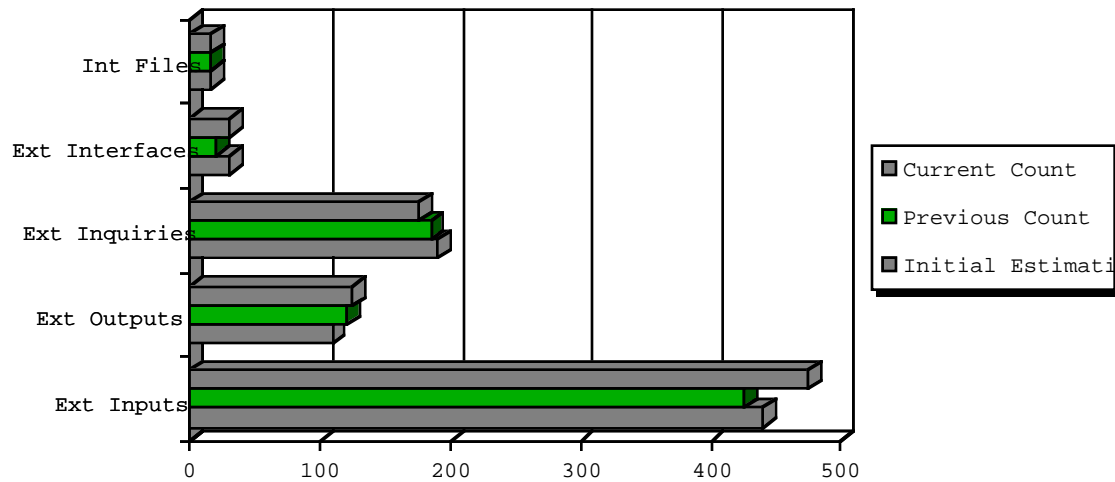
**ACTIONS:**     Detailed explanation from the developing organization and related discussions regarding cost and schedule changes.

**SPECIAL DEFINITIONS:**

Source Line of Code	Instructions created by project personnel and translated into machine code; includes job control language, data declarations, and format statements; it excludes comment statements [IEEE 92];
Estimated new SLOC	Newly developed code;
Estimated reused SLOC	Existing code used as is;
Estimated modified SLOC	Existing code requiring change;
Function Points	number of system externals and internal logical files multiplied by empirically derived weighting functions.

**NOTES:**     Size is an accurate predictor of the resources and the nominal schedule required to develop a product. The principal reasons software project managers experience cost and schedule problems is poor planning [CONSORTIUM 94]. The most frequent cause of poor planning is poor size estimation. A significant amount of the initial code growth and migration can be attributed to estimation errors. Estimation experiences have shown size estimation errors as high as 100 percent [HIHN 91]. More disturbing, reports from the Jet Propulsion Laboratory shows that only 22% of the surveyed professionals actually used size estimates in making cost estimates [HIHN 91]. Consequently, the approach taken in this report focuses on code growth and migration vice quality of initial estimated values. As developers refine design and implement code segments, significant changes (up or down) in code counts are expected. These changes should trigger discussions in related efforts (e.g., test, documentation, quality assurance) and potential schedule milestone adjustments. Ultimately, project managers and development personnel use initial and final code counts in cost models to support future bid rationale and growth discussions.

# Function Point Monthly Report January



## Summary Table

	External Inputs	External Outputs	External Inquires	External Interfaces	Internal Files
Initial Estimation	440	110	190	18	15
Previous Count	425	120	185	20	15
Current Count	475	125	175	31	15
Monthly % Change (+/-)	10%	4%	-5%	55%	0%
Cumulative % Change	8%	13.5%	-8%	72%	0%

Figure 3.1.2-2 Function Point Monthly Report

Function point estimations correlate well with software development schedule and cost data, and unlike SLOC, are language and application independent. Once the SLOC-to-function point ratio for a particular language is determined [HUMPHERY 95], function points can be used to estimate source code size by multiplying the ratio by the number of function points.

Although not as prevalent as SLOC and FP-based sizing estimates, object-oriented sizing metrics are starting to appear more frequently in the literature [KIM 94], [LI 93]. Until models are tuned to correlate object-oriented sizing data with program cost and schedule data, the empirical base necessary to bid efforts is missing. Consequently, object-oriented terms (objects, classes, methods, etc.) are usually translated to equivalent SLOC and FPs for bidding purposes. In the near term, it is still reasonable to use object-oriented sizing data in monthly reviews to measure relative growth in the related areas, to discuss program rationale for growth, and to discuss potential impacts to current cost and schedule estimates.

Examples of object-oriented sizing metrics are found in [CHIDAMBER 94], [LARANJEIRA 90], [LORENZ 94], [WILLIAM 93], and [KIM 94]. Examples are:

- number of scenario scripts [sequence of steps made by the system/user to accomplish a task]
- number of classes
- number of subclasses
- total number of methods in a class (public, private, protected)
- number of methods in a class that are available as services to other classes (public)
- number of instance variables in a class (volume indicator of a class size)

Several object-oriented sizing metrics cited in the literature have cost and schedule implications but are more readily attributed to quality aspects of a system vice estimation. For example, the number of class variables in a class [Lorenz 94] measures the number of localized globals, i.e. common objects for all instances of a class. The number of class variables in a class is certainly correlated with overall program development costs and schedule but more readily relates to the maintainability of a class.

### **3.1.3 Additional Questions**

Software size has a direct impact on the total development cost and a contractor's ability to meet program milestones. Size measurements can be used to help answer the following questions.

- How much do the size estimates change over a period of time during the development process?
- Are the fluctuations stabilizing in the latter development/test phases?
- How does the trend of size estimates and actual data effect the development process?
- Are trends of estimated/actual sizing data being reflected in testing allocations?
- How does the trend of size estimates and actual data effect the resource allocation?
- Is the ratio of new and reused code changing and what impact is this change on estimated schedule and cost?
- Are the changes in size estimates consistent with other collected metrics data (e.g., size vs. cost, size vs. staffing profile size vs. complexity)?

- Are specific CSCIs or software units experiencing more severe fluctuations?
- Are migrations from primary to secondary language (e.g., Ada to C) being reflected in cost/schedule estimations?

Variations in size (increase or decrease) greater than predefined triggering values could indicate:

- Problems in the use or validity of estimating models.
- Problems in the underlying estimation approaches.
- Instability in requirements specifications.
- Problems in understanding the system to be built.
- Unrealistic schedules or productivity.
- Inability to realize expected software reuse benefits.

## **3.2 Software Complexity**

### **3.2.1 Purpose**

The Software Complexity measure tracks the development organization's ability to monitor complexity levels at the Software Unit, and CSCI (Computer Software Configuration Item) level. Major problems that frequently occur with software systems include: 1) the specification of highly complex designs that cannot be readily understood, and 2) the translation of design specifications to code implementations that cannot be adequately tested or maintained. Software Complexity data is collected during Design Inspections (DI), and Code Inspections (CI) points and reviewed for adherence to program established thresholds. Unit-level complexity values are aggregated according to formulas specified below to establish CSCI thresholds. Complexity values exceeding program specified limits or complexity growth above program specified triggering levels require a reduction of complexity of the individual Units, CSCIs or waivers based on project guidance.

### **3.2.2 Description**

#### **3.2.2.1 Functionally - Oriented Complexity Metrics**

METRIC CATEGORY: Quality - Complexity

PURPOSE: Track development organization's ability to maintain an acceptable level of complexity at the Software Unit, and CSCI (Computer Software Configuration Item) level during design, implementation, integration, and maintenance stages.

INPUT DATA: Design/Integration complexity of Software Units / CSCIs

MEASUREMENTS:

- Cyclomatic Complexity
- Essential Complexity
- Module Design Complexity
- Actual Complexity
- Design Complexity
- Integration Complexity
- Global Data Complexity

**TRACKING PERIOD:** Preliminary Design through Post Deployment Software Support for each planned activity

**FREQUENCY OF COLLECTION:** Monthly/Project Reviews

**USAGE:** Data suggests that there is a strong correlation between the complexity of a program segment and the ability of a development organization to adequately test and maintain the program segment. Controlling complexity during the design, implementation, integration, and maintenance phases supports risk reduction goals and reduces life cycle project costs.

Functionally-oriented complexity measures defined in this report are primarily derived from measures defined by McCabe [MCCABE 95]. Specific measurement recommendations are described below.

***CYCLOMATIC COMPLEXITY (CC):*** Cyclomatic Complexity [MCCABE 89] is a measure of the number of independent paths through a program. Cyclomatic Complexity is derived from a flowgraph and is mathematically computed using graph theory. It is found by determining the number of decision statements in a program or program design implementation specification and adding one [ZUSE 91]. For compound decision statements, all Boolean paths are counted.

Although Cyclomatic Complexity produces a quantifiable measurement of complexity, there is no research which has established an absolute complexity threshold for quality software. In one study [WALSH 79], a system contained a total of 276 modules of which half had a Cyclomatic complexity of 10 or less, and half had a complexity greater than 10. The average error rate for the modules in the first group was 4.6 per 100 source statements while the corresponding error rate for the more complex modules was 5.6. Based on research [WALSH 79], [MCCABE 89], many organizations set a complexity triggering value between 7 and 10.

Cyclomatic Complexity at any level is computed via standard complexity toolsets. Complexity toolsets are integrated with standard program design language toolsets, providing the ability to compare complexity values of the implemented code segments with the values estimated in design implementation specifications. Other complexity metrics associated with functionally oriented programming which may be helpful as a measure are briefly summarized in Appendix D.

Figure 3.2.2-1 and figure 3.2.2-2 are examples of a reporting structure providing visibility for Cyclomatic Complexity metrics. In this case, the project upper limits for Average Cyclomatic Complexity were set at 10 for Software Units, and 550 for CSCIs. Cyclomatic values were determined by selecting the 20% (20% level was program selected) most complex Software Units. The supporting table indicates that Software Unit complexity has been managed within the project limits, but the complexity level of the most complex CSCIs still exceeds the thresholds. Based on many project considerations, this may be acceptable behavior. However, the table points to program elements where further discussion/analysis is required. In some instances, based on this analysis, it may be useful to track all Software Units within the critical CSCIs and report complexity data on separate graphs.

***MODULE DESIGN COMPLEXITY (MDC):*** Module Design Complexity measures the amount of interaction between modules in a system. Module Design Complexity quantifies the software integration complexity contributed by a specific Software Unit and is calculated by summing the number of calls received/made by the Software Unit. The more unique calls received or made by a unit, the higher its contribution to software integration complexity.

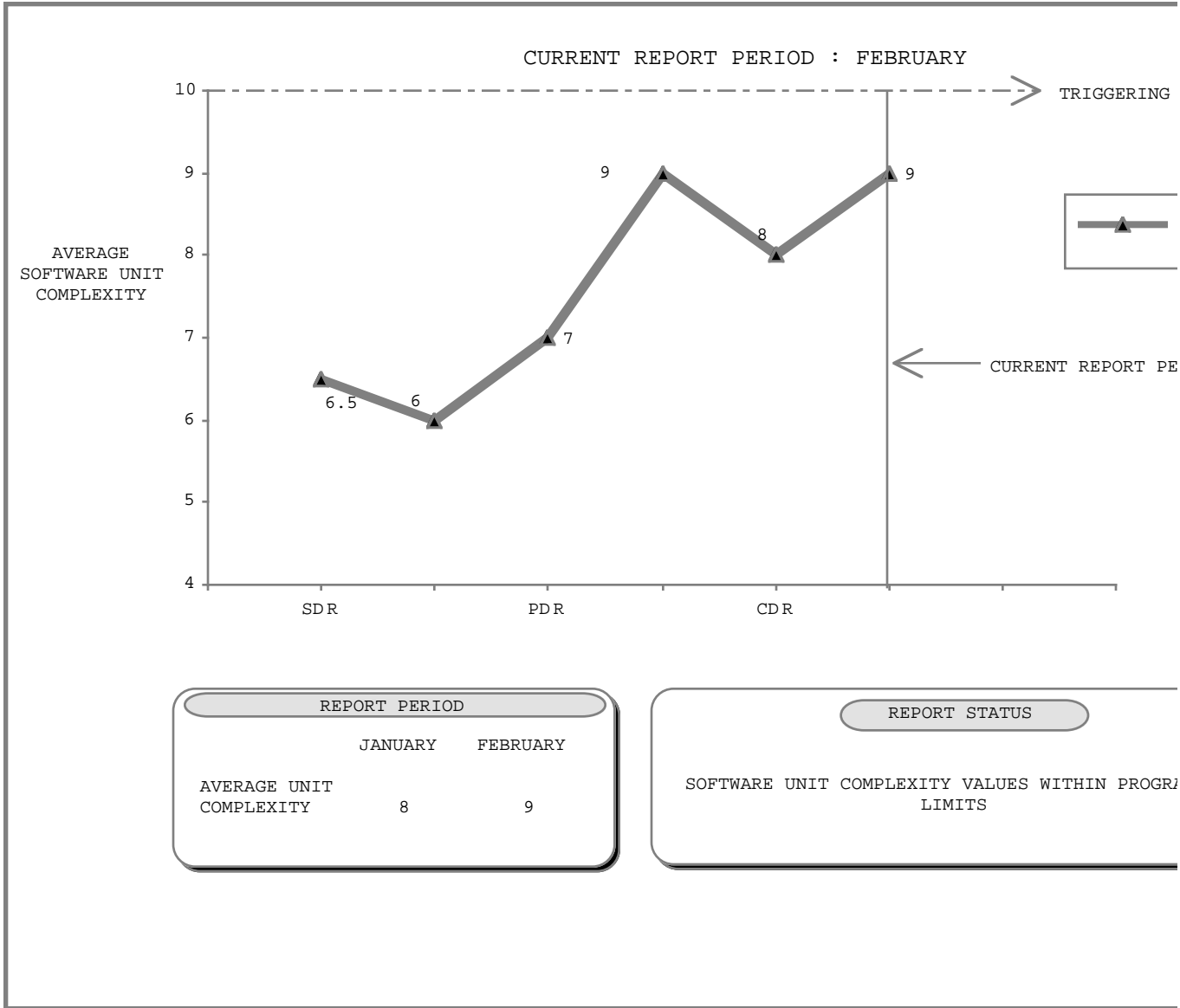


Figure 3.2.2-1 Average Software Unit Complexity

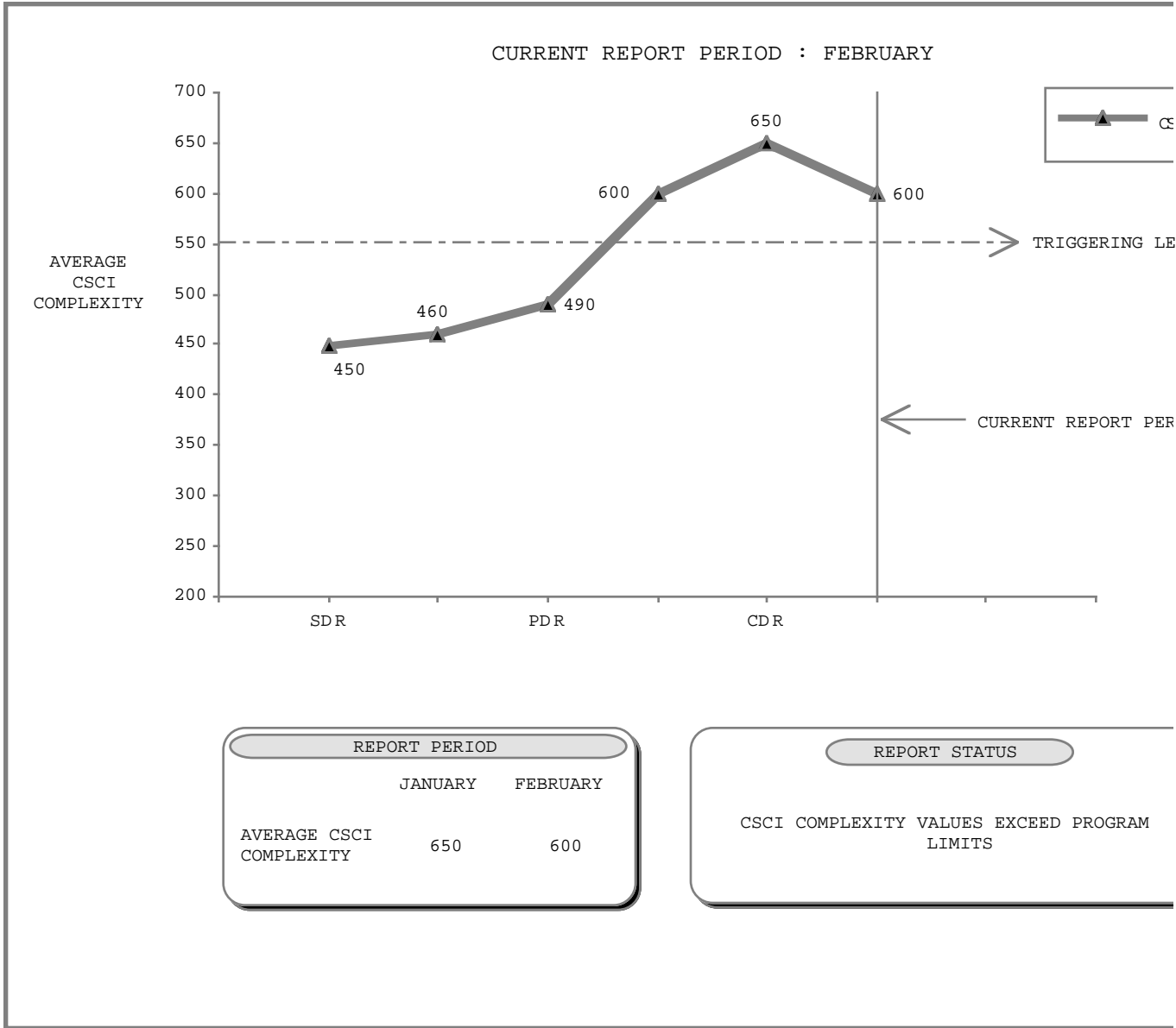


Figure 3.2.2-2 Average CSCI Complexity

Significant monthly or cumulative increases in MDC indicate growth in interface complexity. Although some growth in MDC is to be expected during development, all substantial growth should be questioned during internal reviews. Areas of discussion should include MDC reduction strategies (e.g., movement of public interfaces to private or hidden interfaces) and adequacy of related test fixtures where growth is appropriate.

**DESIGN COMPLEXITY (DC):** While MDC measures the amount of interaction between modules in a system, Design Complexity is the sum of the Module Design Complexity of all Software Units in the software architecture. Design Complexity measures the number of logical paths in a given software architecture and supports the integration complexity calculation and definition of supporting test cases. DC should be reviewed during internal reviews and monitored monthly for any growth beyond program specified triggering levels. Areas of discussion should include specific Software Units contributing to DC growth, potential reduction strategies, and adequacy of related test fixtures.

**ACTUAL COMPLEXITY (AC):** Actual Complexity is the actual number of independent paths tested. It is the number of distinct independent paths traversed during the test phase. Redundant logic 1) increases apparent path complexity above actual path complexity, and 2) reduces efficiency of the testing programs by introducing redundant test structures. Measuring and tracking Actual Complexity improves the quality and maintainability of the software system design by: reducing overall code complexity, identifying areas of potential reuse/streamlining, and reducing the number of test structures necessary to adequately test a code segment. Actual Complexity values should be compared with Cyclomatic Complexity values to probe for areas of redundant code. Additionally, AC measures should be tracked monthly to monitor AC erosion during the development effort. Significant increases during the development of maintenance stages should be reviewed for impact to existing test programs.

**INTEGRATION COMPLEXITY (IC):** Integration Complexity measures the amount of integration testing necessary to adequately test a system. Integration Complexity is the minimum number of unique subtrees of the system software hierarchy. IC values correlate directly with overall testing effort; i.e. increases in IC values require commensurate growth in related test fixtures and provides improved software test effectiveness and efficiency.

**GLOBAL DATA COMPLEXITY (GDC):** Global Data Complexity is calculated by counting the number of accesses to globally specified data regions by Software unit. Global Data Complexity quantifies the complexity of a module's structure as it relates to global data regions. Increasing GDC values indicate a growing dependency on global data structures and should be evaluated for impact on maintenance procedures. Maintenance of such modules is difficult as the side effects of modifying related data regions directly affects a module's behavior. Program development plans should discourage usage of global data region by monitoring GDC values during development. Increases in GDC values should be discussed during internal program reviews and GDC reduction strategies implemented where appropriate (e.g., migration of global data to argument/parameter lists, use of restricted global data regions, etc.).

**TRIGGERING GUIDELINE:** Triggering guidelines or upper control limits (UCL) are provided for CC and EC measures. MDC, AC, DC, IC, and GDC metrics are primarily used to scope testing efforts and identify specific test paths. Since there are no meaningful numerical guidelines for these measures, triggering guidelines focus on growth percentages.

Ten (10) for average Cyclomatic design complexity at Unit level

Four Hundred (400) for average Cyclomatic complexity at the CSCI level

MDC : 5% monthly, 10% cumulative growth

AC : 5% monthly, 10% cumulative growth

DC : 5% monthly, 10% cumulative growth

IC : 5% monthly, 10% cumulative growth

GDC : 5% monthly, 10% cumulative growth

**ACTIONS:** Reduction of Cyclomatic of individual units, CSCIs or waivers based on project guidance

Identification and removal of redundant code structures

Identification and implementation of interface complexity reduction strategies

Review of adequacy of associated test structures and schedules

Reduction of global data region dependencies

**SPECIAL DEFINITIONS:** Cyclomatic Complexity - maximum number of linearly independent paths through a module of code

Software Unit Cyclomatic Complexity - number of linearly independent paths in a Software Unit;

CSCI Cyclomatic Complexity - sum of Unit complexity plus 1;

Module Design Complexity - number of calls made/received by a module

Actual Complexity - actual number of independent paths tested

Design Complexity - sum of MDC of all Software Units in architecture

Integration Complexity - Design Complexity minus the number of unique Software Units plus 1;

Global Data Complexity - number of global data read/write accesses

**NOTES:** Other data suggests strong correlation between the complexity measures and the volume of software changes and the performance of programmers on comprehension, modification, and debugging tasks. Effective management of the complexity data within a program requires an engineering life-cycle that provides visibility for these concerns. An internal review policy, consisting of Design Ready Reviews, Design Inspections, and Code Inspections, is required to provide timely review and corrective action.

Care must be taken when trading off Unit-level implementation complexity and Unit-level interface complexity. Arbitrarily, reducing component complexity by partitioning Software Units into additional Software Units may potentially increase the number of

interfaces (and resulting integration and test efforts). Additionally, other issues need to be considered when evaluating software complexity data [WALSH 79]. Cyclomatic complexity measures are not sensitive to such issues as common block references, external calls, and recursive logic (all of which add to the inherent complexity of a program). Conversely, cyclomatic complexity measurements are overly sensitive to prolonged CASE statements and IF-ELSE IF structures. A comprehensive complexity program balances complexity data with well-defined design/code inspections.

Other problems [SHEPPERD 94] encountered with cyclomatic complexity include: 1) applying structure improving heuristics that lead to an increase in complexity, 2) factoring out of duplicate code in order to increase modularization which can also increase the cyclomatic complexity value and, 3) overlooking factors such as data and functional complexity. It is strongly suggested [GOODMAN 93] that cyclomatic complexity be used more as control gates rather than as actual measures.

### 3.2.2.2 Object-Oriented Complexity Metrics

Object-Oriented methods for requirements analysis, specification, design and implementation provide natural methods for structuring and organizing a model of the problem definition and its solution. Object-Oriented methods lead to software architectures which are more stable, more easily maintained, and more easily reused than systems based on functional and data flow approaches. Key Object-Oriented concepts include:

- Inheritance
- Encapsulation
- Polymorphism

**Inheritance** is defined as the mechanism by which one object acquires characteristics from one or more other objects. **Encapsulation** is a form of data hiding and a technique for minimizing interdependencies among separately written modules by defining strict external interfaces. In effect, encapsulation prevents small systemic changes from having massive ripple effects throughout the system. **Polymorphism** is the ability for an object to belong to more than one classification. Complexity metrics are described below for the inheritance and encapsulation areas as well as cyclomatic complexity metrics applicable to Object-Oriented development.

METRIC CATEGORY: Quality - Complexity

PURPOSE: Track development organization's ability to maintain an acceptable level of complexity at the Software Unit, and CSCI (Computer Software Configuration Item) level during design, implementation, integration, and maintenance stages for object-oriented developments.

INPUT DATA: Design/Integration complexity of Software Units / CSCIs

MEASUREMENTS:

**Inheritance**

- Depth of inheritance
- Fan in
- Number of children

**Encapsulation**

- Cohesion in methods
- Percentage of public/private data

- Access to public data
- Quality**
- Maximum essential complexity
  - Maximum cyclomatic complexity

**TRACKING PERIOD:** Preliminary Design through Post Deployment Software Support for each planned activity

**FREQUENCY OF COLLECTION:** Monthly/Project Reviews

**USAGE:** Object-oriented metrics defined in this report are primarily derived from measures defined by [LORENZ 94] and [MCCABE 95]. Recommendations for specific measurement are described below. Additional object-oriented complexity metrics are described in [CHIDAMBER 94] and Appendix D.

***INHERITANCE:*** The Depth of Inheritance (*DOI*) metric measures the position of a class in an inheritance hierarchy or how many ancestor classes can potentially affect this class. Interpretation of the Depth of Inheritance measure requires a trade-off between the increase in complexity derived through inheritance of functionality and data and the complexity-reduction derived from program reuse. The deeper a class is in the inheritance hierarchy, the greater the number of methods it will inherit, making it more complex, harder to test, and harder to maintain. But as reusability of a class increases, the value of complexity decreases gradually because inheritance permits code reusability.

Fan In is the number of classes from which a class is derived. High values indicate excessive use of multiple inheritance. Systems exploiting multiple inheritance are inherently harder to test, maintain (side effects) and expand.

Number of children (*NOC*) is the number of classes derived from specified parent class and/or the number of immediate subclasses subordinate to a class in a class hierarchy. When a large NOC exists, it can cause increased difficulty in understanding the relationships among the objects in a class and lead to increased maintenance cost. The value in measuring NOC is that NOC indicates the number of classes which will be directly affected by a change to the parent.

Another factor to consider during object-oriented development is the trade-off between effective use of inheritance mechanisms and object coupling. Since inheritance promotes software reuse in an Object-Oriented paradigm, it also creates possibilities of violating encapsulation and information hiding.

***ENCAPSULATION:*** Encapsulation is the packaging or binding together of related items. Objects encapsulate their data, making them accessible to other modules through messages or well defined methods. Encapsulation improves module cohesion and minimizes the interdependence among separately written modules through abstract interfaces. Objects with low cohesion are more likely to be changed and are more likely to have undesirable side effects when they are changed.

The complexity of program is related to number of classes and the number of classes is related to the degree of reuse of class. The larger the number of classes is, the greater the efforts of the programmer and the less the degree of reuse of class is. Cohesion of a class is characterized by how closely local methods are related to local instance variables in the class. Class size tends to decrease with increasing cohesion of the method in the class.

The Lack of Cohesion In Methods (*LOCM*) metric measures the lack of cohesion of a class. It is defined as the percentage of methods in a class that do not access a specific data field, averaged over all data fields in the class.

Percentage of Public and Protected Data (*PPPD*) is the percentage of data in a class that is public/protected. *PPPD* also shows how much of the data within a class is available to other objects.

Access to Public Data (*APD*) is the number of times that a classes' public/protected data is accessed. *APD* measures the impact of data that is not controlled by a class.

***QUALITY:*** Quality metrics focus on the O-O extensions to standard cyclomatic complexity measures. Class-oriented cyclomatic complexity is defined as the maximum cyclomatic complexity of the methods in a class. Definition and usage of this measure is described in 3.2.2.1. Maximum values are selected to establish timely discussion of class-level complexity issues. In practice, it may be more beneficial to use class-average cyclomatic.

**TRIGGERING GUIDELINE:** Triggering guidelines or upper control limits are provided for maximum (CC). Guidelines levels based on empirical evidence are not currently available for the other object-oriented complexity measures. Programs should monitor growth levels monthly and use 5% triggering levels as a mechanism to review complexity issues.

Ten (10) for maximum (CC)

**ACTIONS:** Reduction of complexity of individual Units, CSCIs or waivers based on project guidance

**SPECIAL DEFINITIONS:**

class - a template, pattern, or blueprint for a category of structurally identical items (common behavior, common relationships, common semantics);

class hierarchy nesting level - depth of a class in an inheritance hierarchy

encapsulation - a modeling and implementation technique that separates the external aspects of an object from the internal, implementation details of the object (also called information hiding)

fan in - count of the number of other components that can call or pass control to a component

inheritance - mechanism where one object acquires the characteristics (attributes and operations) from one or more other objects

number of children (NOC) - number of immediate specializations in the form of classes and subclasses

object - a model of real-world physical and conceptual things

object cohesion - a measure of how logically related the components of the internal view of an object are to each other

object coupling - describes the degree of interrelationships among the objects that make up the system

object-oriented - a development technique that uses objects as a basis for analysis, design, implementation

polymorphism - the property that an operation may behave differently on different classes

public - an attribute or operation accessible by methods of any class

private - an attribute or operation accessible by methods of the current class only

**NOTES:** The metrics described above are only a subset of the O-O metrics described in the literature. Other metrics relevant to the Object-Oriented paradigm are briefly described in Appendix A.

### 3.2.3 Additional Questions

Software complexity has a direct impact on a contractor's ability to adequately test a system. Controlling complexity supports program risk reduction goals and has the potential to significantly reduce maintenance costs. Complexity measurements can be used to help answer the following questions:

- Are test plans/procedures adequate to exercise program segments?
- Are expected problem report densities reflective of recorded program complexities?
- Are specific CSCIs/ Software Units inherently more complex, thus requiring more program visibility (schedules, specifications, test structures, etc.)?
- Are aggregated CSCI complexity values significant enough to suggest further logical decomposition of software entities?
- Are inheritance structures too extensive to effectively maintain?
- Do inheritance structures promote unexpected side-effects during maintenance stages?
- Are modules sufficiently cohesive to support reuse and minimize side effects?
- Are data elements sufficiently protected through proper levels of encapsulation?

Variations in size greater than predefined triggering levels may lead to :

- Inadequately tested Units.
- Excessive growth in maintenance costs.
- SPR density not consistent with unit expectations.
- Integration and test schedule / cost growth.

- Increased program risk.

### 3.3 Software Schedule

#### 3.3.1 Purpose

The Software Schedule performance measures provide CORs and development and post deployment software support personnel with a comparison of actual milestones completed against established milestone commitments. These measurements quantify the contractor's performance toward meeting commitments for delivering products and completing milestones. The Software Schedule measure tracks the development organization's ability to maintain the software development schedule by tracking the delivery of software packages as defined in the program Work Breakdown Structure (WBS). Estimated schedules are derived by adjusting current program schedules by a weighting factor based on actual cost of work performed (ACWP) vice budgeted cost of work scheduled (BCWS).

#### 3.3.2 Description

METRIC CATEGORY: Progress - Schedule

PURPOSE: Track development organization's ability to maintain the software development schedule by tracking the delivery of planned software work packages.

INPUT DATA: Actual Cost of Work Performed (ACWP), Budgeted Cost of Work Scheduled (BCWS), Current Months in Project Schedule

TRACKING PERIOD: Full Life-cycle for each planned activity

FREQUENCY OF COLLECTION: Monthly/Major Milestone Reviews

USAGE: Use of costing data to estimate schedule progress requires close coordination between costing staff and engineering/project management. Care must be taken in the use of cost data to derive schedule information. Confirmation of work performed can be determined by correlating BCWP data with development, testing, and incremental build measurement data. Claiming completion of project milestones requires a well-defined Work Breakdown Structure and associated data dictionary, objective entry and exit criteria for each event and activity specified in the WBS, and additional progress metrics (design, development, test, etc.) to substantiate work performed.

Figure 3.3.2-1 provides a sample schedule graph output. During month SRR, the plot indicates that given the current rate of productivity, project completion will require approximately 22 months instead of the originally scheduled 18 months. This was determined by calculating the ratio of worked performed to work scheduled and applying this multiplier to adjust the project schedule. The chart provides data for the January and February time frames where February is the current reporting month. January's data indicates no improvement in productivity with an approximate four-month schedule slip still projected (18 months / % work done / % \$ expended). However, February data suggests continued erosion of productivity with 45% work done and 60% of budgeted funds expended, and a resulting schedule of approximately 25 months. At this point, the schedule was revised and extended to 22 months, indicating a 4 month slip from the original schedule.

Plots of this form can be used to identify current schedule estimates and extrapolate on schedule trends. Although minor perturbations are expected, positive slopes above the

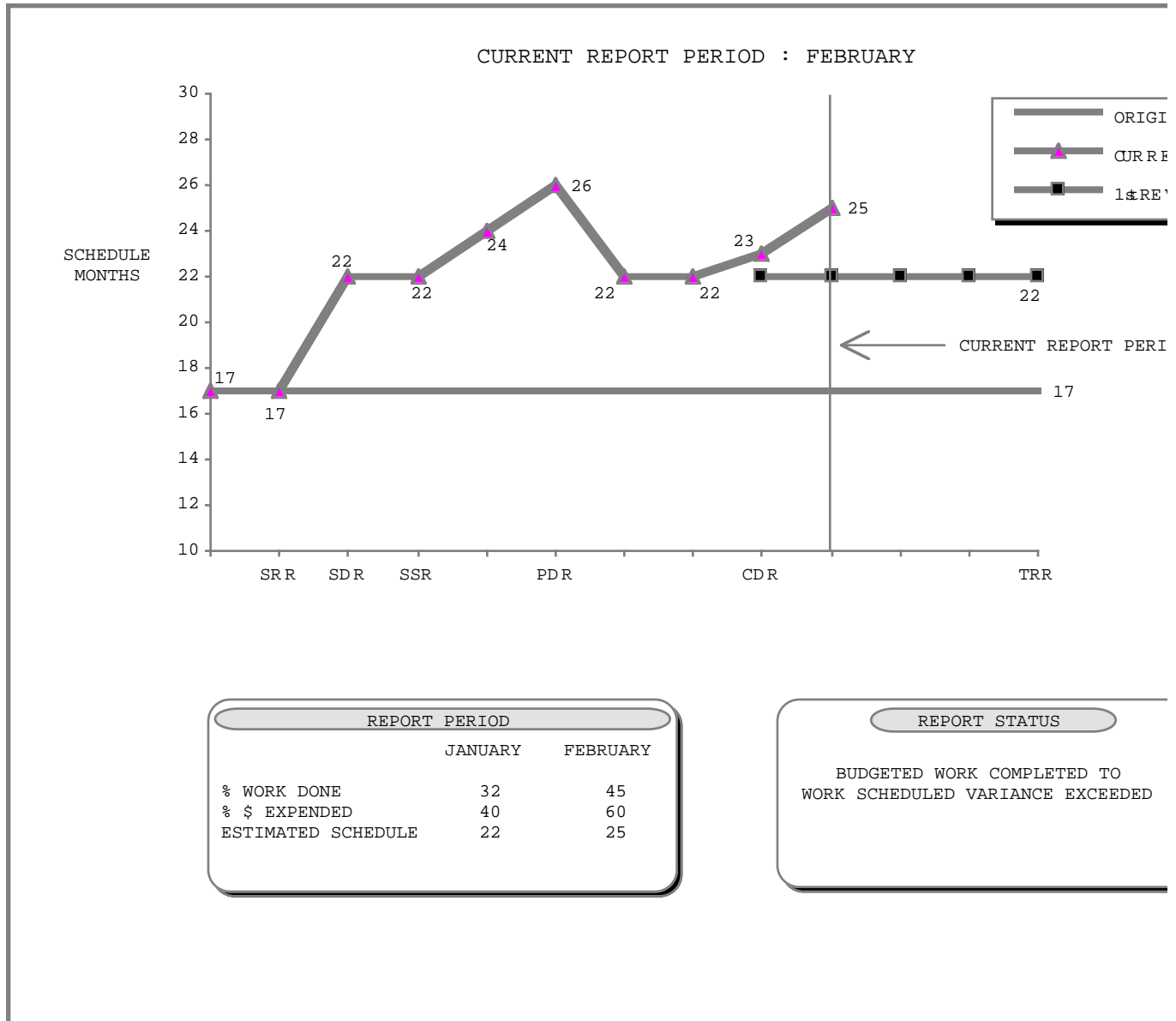


Figure 3.3.2-1 Software Schedule

triggering levels indicate significant risks to ultimate project schedules while negative slopes indicate productivity control. Although it is not clear at what point it becomes necessary to formally adjust project schedules, setting triggering levels of 5% for monthly changes in estimated schedule and 10% for cumulative changes in estimated schedule provides a point from which to begin discussions. For more critical CSCIs or for WBS elements on the critical path, it may be necessary to establish more stringent values and track on these elements with separate graphs.

To determine the stability and progress of the schedule, a summary table [ROZUM 92] in which for each revision, the date, time since last revision, and schedule change (acceleration or slip) is recorded may be useful.

**TRIGGERING GUIDELINE:**     5% from previous month estimates;  
                                         10% cumulative

**ACTIONS:**     Detailed explanation from the developing organization and related discussions regarding schedule improvements;

**SPECIAL DEFINITIONS:**     Budgeted Cost of Work Scheduled - cost estimates for individual work packages as determined by engineering inputs and grouped via a Work Breakdown Structure;

                                         Actual Cost of Work Performed - cost actuals for individual work packages completed as determined by monthly cost accounting and Work Breakdown Structure;

                                         Estimated Schedule - Project Schedule(new or revised) divided by (ACWP/BCWS)

**NOTES:**     Effective use of Software Schedule metrics requires detailed Work Breakdown Structure and a cost accounting system that allows detailed milestones to be planned and costing data associated with claimed milestones to be collected.

### **3.3.3 Additional Questions**

Software Schedule measurements can be used to help answer the following questions.

- Are there process activities that are routinely completed late? (Correlate with development progress metrics.)
- Are the schedule/adjusted schedules consistent with each other and with the Program Plan and the Software Development Plan?
- Is the number of schedule slips increasing or decreasing through program phases?

Variations in schedule greater than predetermined triggering values could indicate:

- Problems in the use or validity of estimating models.
- Unrealistic schedules or productivity estimates.
- Correct spending profile (BCWS) but significant staff turnover rate.
- Instability in requirements specifications.

- Problems in understanding the system to be built.
- Instability of development environment ( tools, platforms, etc.).

## 3.4 Software Development

### 3.4.1 Purpose

The Software Development performance measurements provide CORs and development and post deployment software support personnel with a quantitative indication of Software Unit-level work progress. The measurement uses data on the planned and actual progress of software Unit-level development ( # of Software Units designed, # of Software Units coded/tested, # of Software Units integrated) to assess whether an activity is complete and the contractor is ready to proceed to successive activities.

Early insight into deviations from planned Unit-level progress is essential for early corrective action to be taken. Effective use of these measures requires that the contractor 1) use a detailed level of planning, and 2) define specific entry/exit criteria for each of work activities being tracked.

### 3.4.2 Description

METRIC CATEGORY: Progress - Development

PURPOSE: Measure development organization's ability to keep Software Units design, coding, test, and integration activities on schedule.

INPUT DATA: Planned number of Software Units to be designed each month  
Actual number of Software Units designed

Planned number of Software Units to be coded/tested each month  
Actual number of Software Units coded/tested

Planned number of Software Units to be integrated each month  
Actual number of Software Units integrated

TRACKING PERIOD: Preliminary Design through Post Deployment Software Support

FREQUENCY OF COLLECTION: Monthly

USAGE: Associated plots of Software Units designed, coded/tested, and integrated should show consistent, positive slope. Figure 3.4.2-1 contains a sample graph providing visibility for Software Unit development progress. Software Unit design begins at or around the Preliminary Design Phase and continues through the Critical Design Phase. Identification of the number of Units successfully completing design activities is verified through internal reviews. Figure 3.4.2-1 indicates that this process was managed effectively until just before CDR. At that time, the number of Software Units actually designed deviates significantly from the number planned. Several potential reasons cited for this diversion from planned design schedules were staff turn-over on key Software Units and evolving requirements. Because for the affected Software Units, code completion and test was not scheduled until the latter part of the Implementation phase, this difference did not immediately translate into Software Unit code and test variances. Figure 3.4.2-1 also indicates that during the current reporting period, the number of Software Units actually integrated differs significantly from

the number planned. The tabular data details that this difference exceeds the 10% window defined for the project. Slips at this stage of the development probably indicate that the overall project schedule cannot be maintained. Correlating this data with other project metrics, may lead the monitoring organization to effect schedule improvements with the contractor.

Based on the complexity of the project, it may be necessary to plot very large CSCIs separately, or gather defect density (Section 3.7) information to provide more visibility into any schedule perturbations. Triggering levels may need to be variable, becoming more stringent during the integration and test phases. Because an understanding of the real impact of exceeding project specific windows for Software Unit design, code and unit test, or integration counts may be lost by tracking only summary counts, it may be more appropriate to focus on Software Units with the potential to more adversely affect schedules. These Software Units may include those with the largest number of SLOCs, highest complexity, or highest associated BCWP.

**TRIGGERING GUIDELINE:** 10 - 20% difference between planned/actual; % based on the number of required Software Units

**ACTIONS:** Detailed explanation from the developing organization and related discussions regarding cost and schedule improvements.

**SPECIAL DEFINITIONS:** None

**NOTES:** Detailed data needed to generate progress graphs is reviewed at the Unit Design Ready Reviews, Design Inspections, Code Inspections, and Integration Ready Reviews [MIL-STD-498]. These reviews also provide the opportunity to collect quality-oriented data such as defect density from errors found at design inspections, defect density from errors found at test inspections, etc.

Equivalently, for object-oriented development, tracking the number of objects/methods designed, developed, and integrated provides early visibility into potential development schedule problems. For a large-scale, MIL-STD-498 compatible development, the relationship between objects and units and classes and CSCIs may be many to one. Consequently, monthly reports may focus on a particular software unit and the related objects, with triggering levels tailored to the specific unit [Figure 3.4.2-2]. Since the number of objects may be extensive for a large system, in general triggering levels should be narrowed to at most 10%.

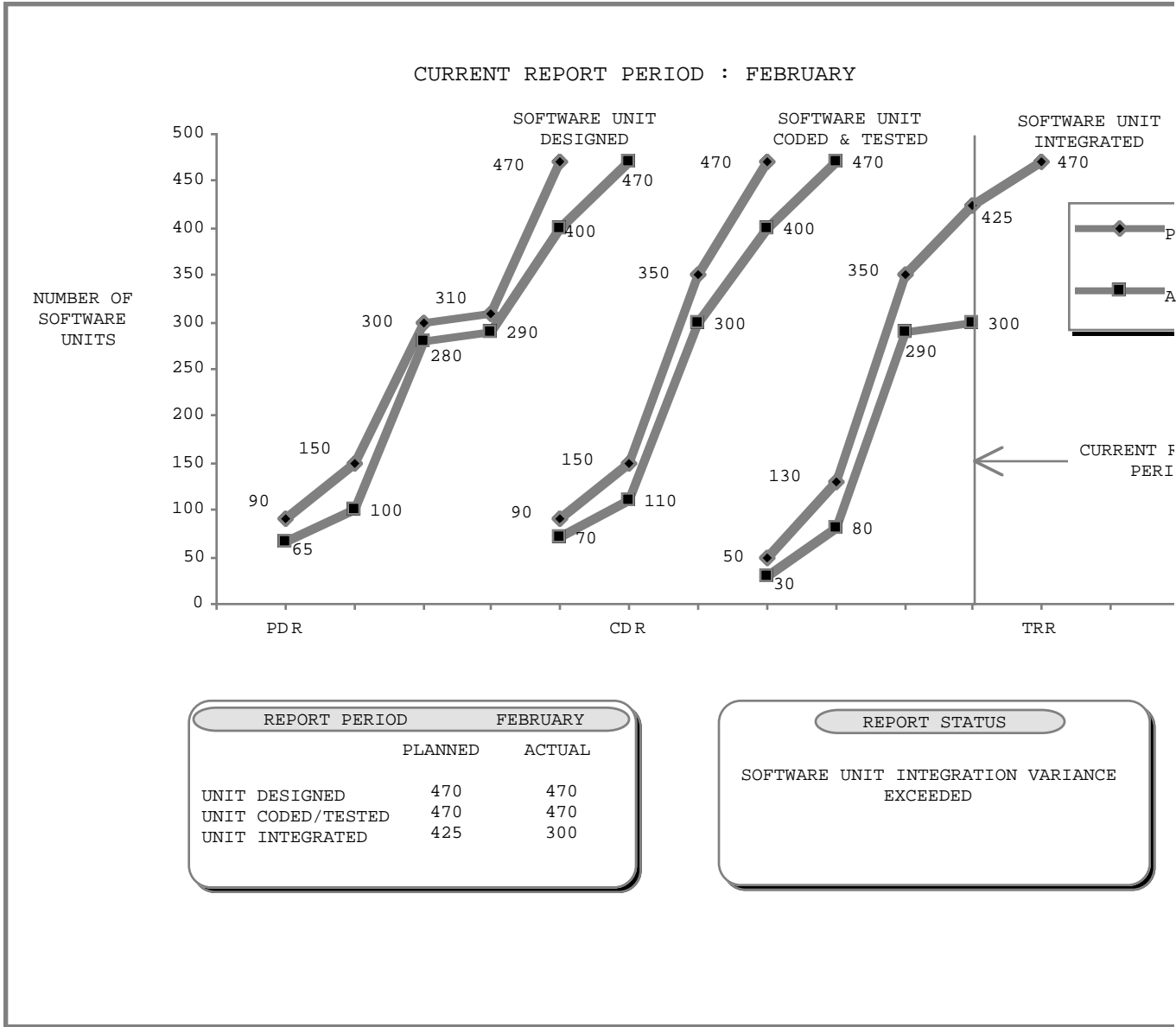


Figure 3.4.2-1 Software Development Progress

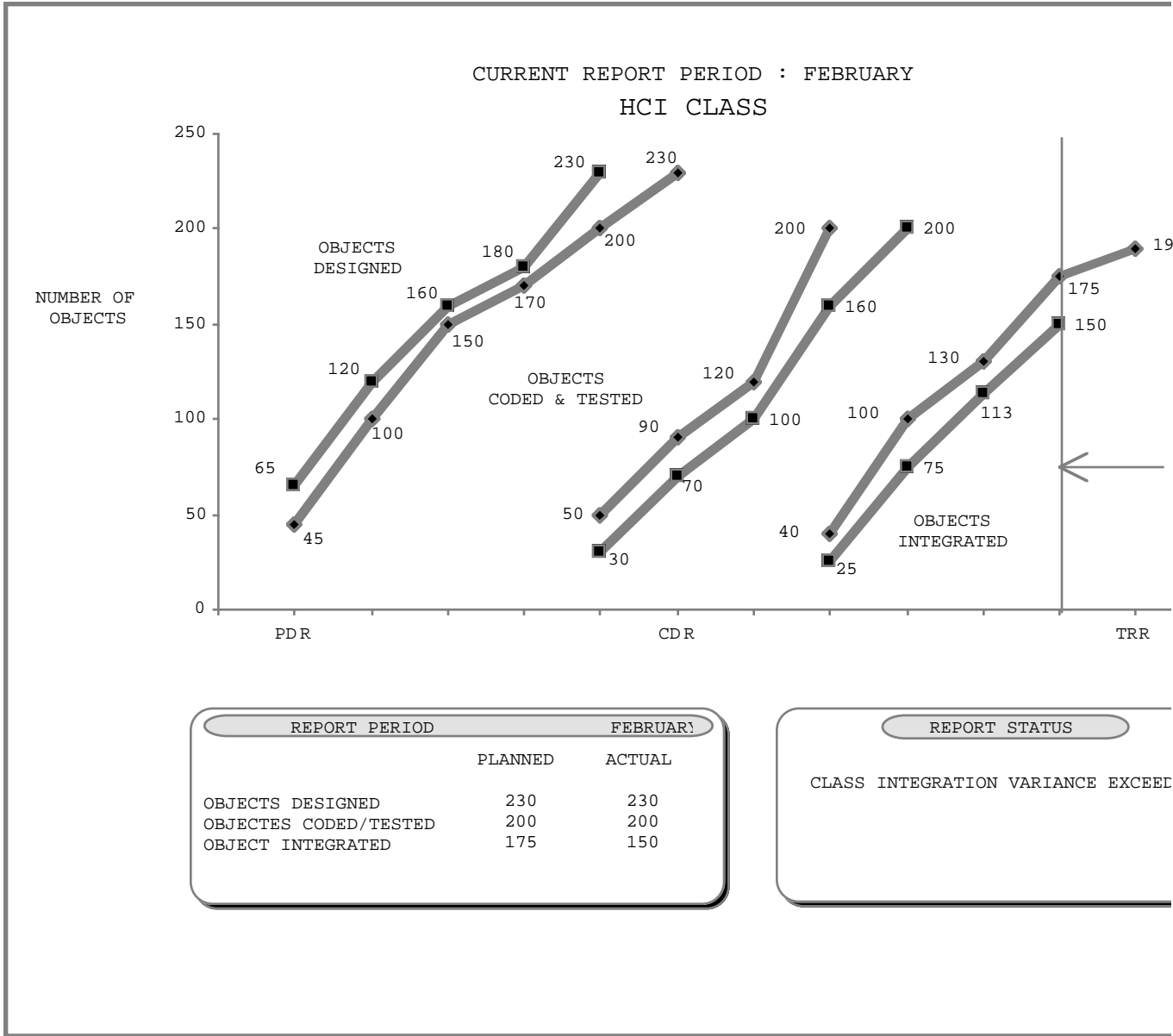


Figure 3.4.2-2 Software Development Progress (Object-Oriented)

Other examples where collection of software development-oriented metrics might prove useful include [AIRF 86], [DECKER 91], [LANDIS 90], [BETZ 91]:

- number of software requirements in the system-level specifications that are documented in the Software Requirements Specifications (SRS).
- number of software requirements in the system-level specifications that are documented in the Interface Requirements Specifications (IRS).
- number of SRS and IRS requirements documented in the Software Design Document (SDD).
- number of SRS and IRS requirements documented in the Interface Design Document (IDD).
- number of Software Units with Program Design Language (PDL) completed.
- number of CSCI integration and test procedures completed.
- number of expert system rules designed, developed, tested.
- number of logical data elements defined.
- number of physical data elements defined.
- number of high order language elements developed (e.g., SQL statements, UNIX scripts).

### 3.4.3 Additional Questions

To assess whether the planned software development completion dates are achievable the COR / Program Manager should [ROZUM 92]:

- Ensure that the completion and exit criteria for software development milestones are well-defined and verifiable (e.g., for code inspections, the internal review is complete as determined by Software Unit Development Folder signatures, all action items are closed, and the source code is under configuration management control). If they are not well-defined, then true progress may be disguised by having work reported as completed when work still remains.
- Take into account the quality of the completed work items, especially during the requirements and design phases where problems are less costly to correct [BOEHM 81]. A rough judgment of the quality during these early stages can be obtained by correlating development progress with quality progress measures. If quality is questionable, then the COR/Program Manager should consider delaying some early milestones to ensure a higher quality product.
- Extrapolate the rate of progress (the slope of the actual completed work item curve between the current report and the last report) to determine if the trend is converging toward or diverging from the planned completed work item curve. If the actual curve is significantly lower than the planned curve and the slope indicates further erosion, this is a cause for concern.

Software Development measurements can be used to help answer the following questions concerning an engineering process:

- Are there software development processes (e.g., Design Inspections, Code Inspections, etc.) that are routinely completed late?

- Are the schedule/adjusted schedules consistent with each other and with the Program Plan, Software Requirements Specifications, and the Software Development Plan?
- Is the number of schedule slips increasing or decreasing over time?
- Does the actual performance of the software development indicate schedule slippage and the necessity of a replanning effort at the program level?
- Do triggering levels need to be adjusted for specific CSCIs or Software Units based on current trends?
- Are specific units affecting overall program schedules and milestones (e.g., poorly specified unit interface impacting integration and test schedule)?

Software Development measurements indicating significant (greater than predefined triggering levels) differences between estimated and actual values could indicate:

- Instability in requirements specifications.
- Problems in understanding the system to be built.
- Instability of development environment ( tools, platforms, etc.).
- Inadequate Unit-level review processes.
- Higher than expected test failure rates (correlate with action item counts, SPR density measures, etc.).
- Significant staffing profile turnover.
- Immaturity of contractor with specific development processes (e.g., object-oriented vice functional).
- Immaturity of COTS products associated with particular software units.
- Inability to capitalize on level of reuse estimated.

## 3.5 Software Testing

### 3.5.1 Purpose

Software Testing measurements provide Contracting Office Representative(CORs) and development and post deployment software support personnel with an indication of testing progress. The planned number of CSCI/System test procedures to be executed is initially recorded in the contractor's software test plans and procedures documentation. Subsequently, these estimates are updated monthly during the CSCI/System integration and test phases to compare actuals with the estimated values.

### 3.5.2 Description

METRIC CATEGORY: Progress - Testing

PURPOSE: Measure development organization's ability to maintain testing progress.

INPUT DATA: Planned number of CSCI tests to be completed each month  
Actual number of CSCI tests to be completed each month

Planned number of system tests to be completed each month  
Actual number of system tests to be completed each month

TRACKING PERIOD: CSCI testing (TRR) through System Testing (PCA) for each planned activity

FREQUENCY OF COLLECTION: Monthly

USAGE: Many projects experience a fair number of failed tests, rework, and associated schedule perturbations. The extent of the differences between scheduled and completed tests provide indicators of the CSCI readiness for system testing. Figure 3.5.2-1 is an example of a reporting structure providing visibility for CSCI/System test progress. The monthly tabular data indicates that 440 CSCI-level test threads were planned but only 340 were successfully completed. Based on 10% triggering levels between planned and actual, a CSCI variance report was generated. Based on the proximity of this report period to the start of System-level testing, it is doubtful that system-level testing can proceed as planned.

Related issues that surface when interpreting test progress measurement data are the efficiency and effectiveness of the testing program. Test efficiency is a measure of the productivity of the testing program. Test effectiveness is a measure of the quality of the results obtained by the testing program.

Test efficiency is usually measured as a quotient of sizing data and testing effort. Examples of potentially useful efficiency metrics include: number of function points / testing effort, number of requirements / testing effort, number of rules / testing effort, and number of scripts / testing effort. Test efficiency is primarily used as a historical data point to validate test schedules and milestones or to support test effort analysis. As program sizing data is updated during program development, test efficiency data can be used to scope impacts to the testing effort and determine the realism of resulting schedule modifications. Other efficiency measures focus on defect identification and the amount of effort required to uncover defects. Although test stage efforts are relatively fixed based on SLOC/model estimates, the magnitude of the regression testing program and ultimately the decision to turn over tested components to the next level of integration testing or to deployments teams should include defect/cost decisions. Customer/contractor test reviews should include analysis of defects to cost-to-uncover ratios to determine program value of continued testing.

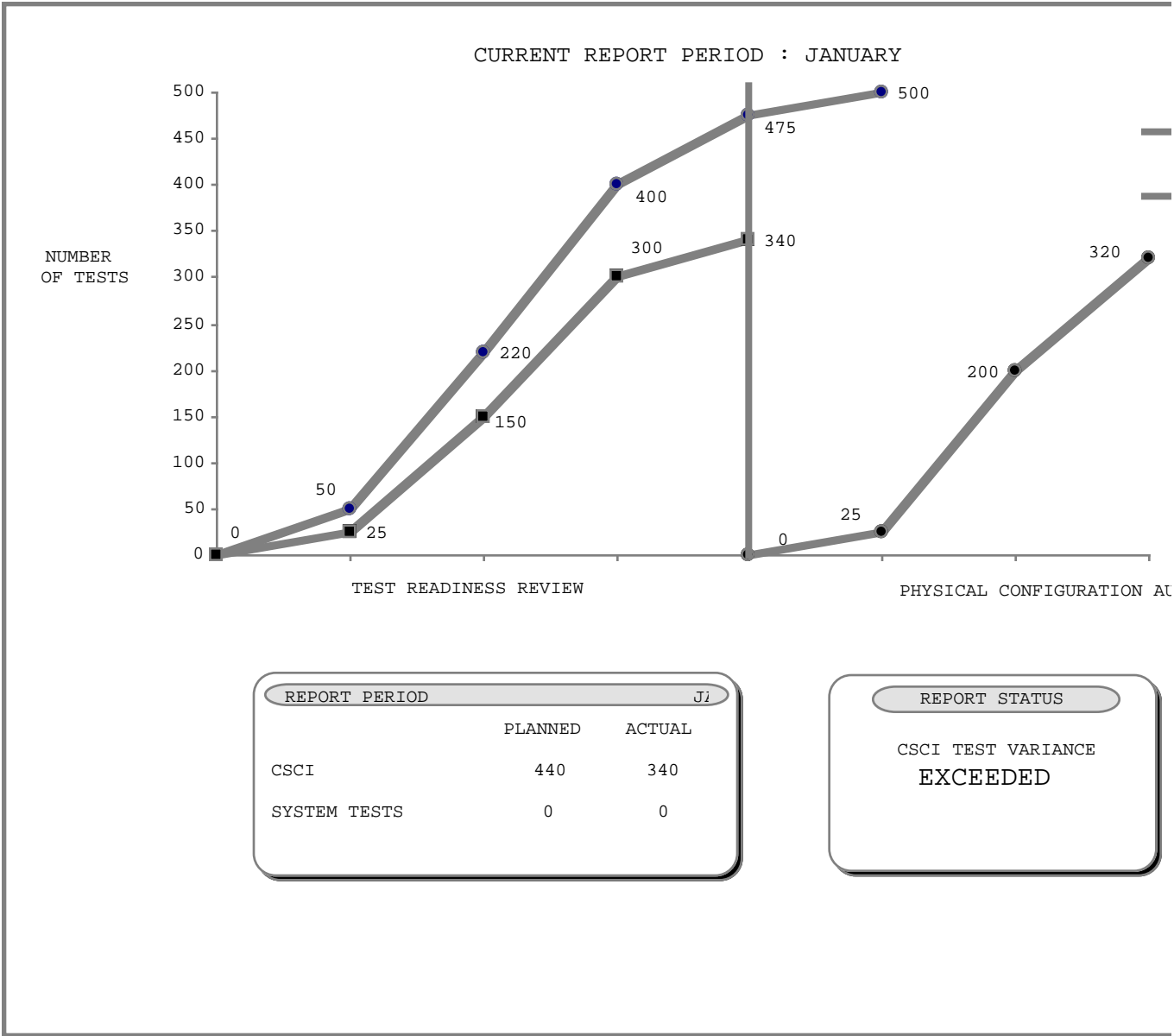


Figure 3.5.2-1 Software Test Progress

Test effectiveness measures are customer-oriented and focus on defects found during PDSS stages. One example of a test effectiveness measure is:

$$\frac{(\text{\#defects uncovered during test}) + (\text{\#defects uncovered during PDSS})}{(\text{\#defects uncovered during test})}$$

Test efficiency measures may be partitioned according to severity using MIL-STD-498 guidelines for leveling.

**TRIGGERING GUIDELINE:** 10% difference from planned and actual test completion

**ACTIONS:** Reallocation of resources to address testing deficiencies or reschedule of system-level test efforts.

**SPECIAL DEFINITIONS:** None

**NOTES:** CSCI and System-level tests are detailed in Software Test Description and System Acceptance and Test Procedures or equivalent documentation structures.

For a large-scale, MIL-STD-498 compatible development the relationship between objects and units, and ultimately classes and CSCIs may be many-to-one. Object oriented developments may require adjustment of measurements/graphs to provide visibility to the number of test cases allocated to objects, classes, or subclasses.

### 3.5.3 Additional Questions

Software Testing measurements can be used to help answer the following questions:

- Are the schedule/adjusted testing schedules consistent with each other and with the Program Plan and Software Development Plan?
- Are the number of schedule slips increasing or decreasing through CSCI/System test phases?
- Does the actual testing performance indicate schedule slippage and the necessity of a replanning effort at the program level (schedule and resources)?
- Do graphs of efficiency data indicate potential schedule and cost problems ?
- Can reworked testing schedules be validated with empirically-based efficiency data?
- Do graphs of defects / cost-to-uncover ratios indicate potential for product turnovers?
- Do triggering levels need to be adjusted for specific CSCIs/System testing based on current trend?

Software Testing measurements indicating significant (greater than predefined triggering levels) differences between estimated and actual values could indicate:

- SPR densities greater than expected (correlate with density measurements).
- Instability of test suite (tools, platforms, etc.).
- Inefficiency of regression testing procedures.

- Complexity levels for CSCIs higher than project established standards (correlate with complexity measurement).
- Unit/CSCI interfaces not properly used or specified.

## 3.6 Software Builds

### 3.6.1 Purpose

Software Build measurements provide CORs and development and post deployment software support personnel with an indicator to track the schedule and the number of Units per release in order to monitor a contractor's ability to preserve schedule and functionality in each release. The planned number of Software Units to be integrated in each build is initially recorded in the contractor's Software Development Plan and Software Integration and Test Plan. Subsequently, these estimates are updated monthly during the Unit integration and test phase to compare actuals with the estimated values.

### 3.6.2 Description

METRIC CATEGORY: Progress - Software Builds

PURPOSE: Monitors development organization's ability to maintain incremental release schedule by integrating Software Units.

INPUT DATA: Number of tested Units per build planned  
Number of tested Units per build actual

TRACKING PERIOD: Software Unit Integration and Test phase for each planned activity

FREQUENCY OF COLLECTION: Monthly

USAGE: Many projects experience a significant amount of schedule perturbation at the Unit-integration level; the tendency is to begin the next level of release testing before adequate closure of the previous build or to migrate Software Unit functionality to the next release. Tracking planned versus actuals provides visibility into this level of testing and replanning based on established trigger points. Figure 3.6.2-1 is an example of a reporting structure providing visibility for build progress. The planned number of Software Units to be integrated for each release is indicated on the ordinate axis and the release date is shown by corresponding abscissa coordinate. Each month the actual number of Software Units integrated is reported and differences greater than the triggering value are highlighted. Figure 3.6.2-1 reveals that during the current reporting period (February), Increment 2 and 3 integration goals are lagging significantly behind schedule ( 325 actual vice 375 planned and 325 actual vice 425 planned). Although the Software Unit releases are internal configuration items, the potential for CSCI/System-level schedule slips exists. Corrective action may include reallocation of resources, adjustment of project-level testing milestones, or insertion of another build (1A) to accommodate continued testing of increment 2.

TRIGGERING GUIDELINE: 10 - 20% difference between actual and planned integrated Software Unit

ACTIONS: Detailed report indicating Unit integration problem areas; reallocation of resources to address testing deficiencies or reschedule of subsequent incremental build test efforts; identification of any Unit migration among releases.

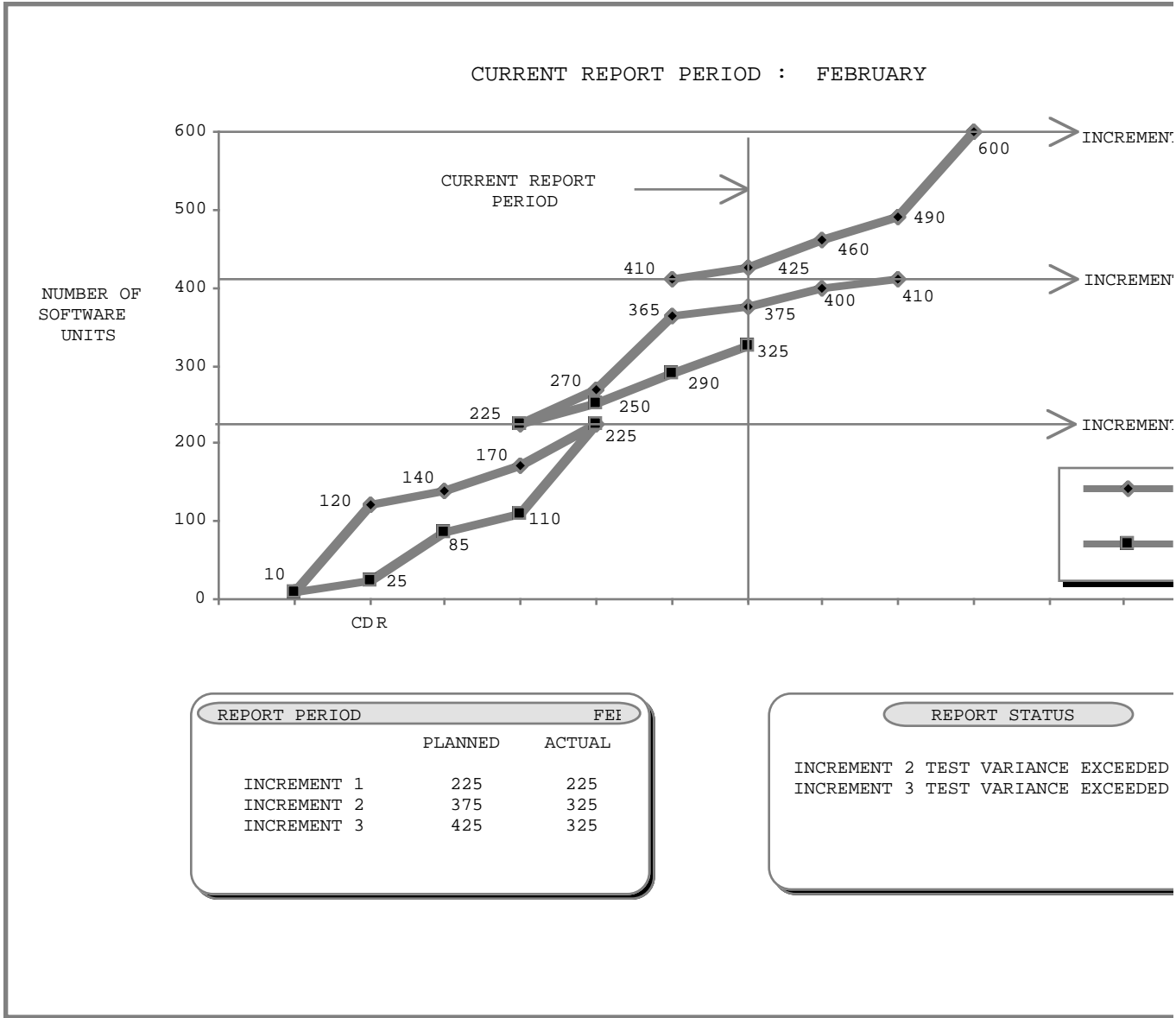


Figure 3.6.2-1 Software Build Progress

**SPECIAL DEFINITIONS:** None

**NOTES:** Object-oriented development may require adjustment of measurements/graphs to provide visibility to integration of objects within a unit, integration of objects within an increment, or class structures allocated to increments. Figure 3.6.2-2 provides an example of an object-oriented reporting structure for Software Builds.

### **3.6.3 Additional Questions**

Software Build measurements can be used to help answer the following questions:

- Are the schedule/adjusted Software Unit integration and test schedules consistent with each other and with the Program Plan, Software Development Plan , and Software Integration and Test Plan?
- Are the number of schedule slips increasing or decreasing through Unit build releases?
- Does the actual integration testing performance indicate schedule slippage and the necessity of a replanning effort at the program level (CSCI / System test schedule and resources)?
- Do triggering levels need to be adjusted for specific CSCI integration testing based on the current trend?

Software Build measurements indicating significant (greater than predefined triggering levels) differences between estimated and actual values could indicate:

- SPR densities greater than expected (correlate with density measurements).
- Instability of test suite (tools, platforms, etc.).
- Inefficiency of regression testing procedures.
- Inability to capitalize or reuse within class/structures.
- Complexity levels for Software Units/classes/CSCIs higher than project established standards (correlate with complexity measurement).
- Software Unit interfaces not properly used or specified.

## **3.7 Software Defect Reporting**

### **3.7.1 Purpose**

Software Defect measurements provide CORs and development and post deployment software support personnel with an indication of the quality of the test program and the tested components. It provides managers and development personnel with information on the readiness of the product to proceed to the next stage of testing or suitability for release. It supports analysis of the underlying processes used to develop a product through root analysis of problem report forms.

### **3.7.2 Description**

**METRIC CATEGORY:** Quality - Software Defect Reporting

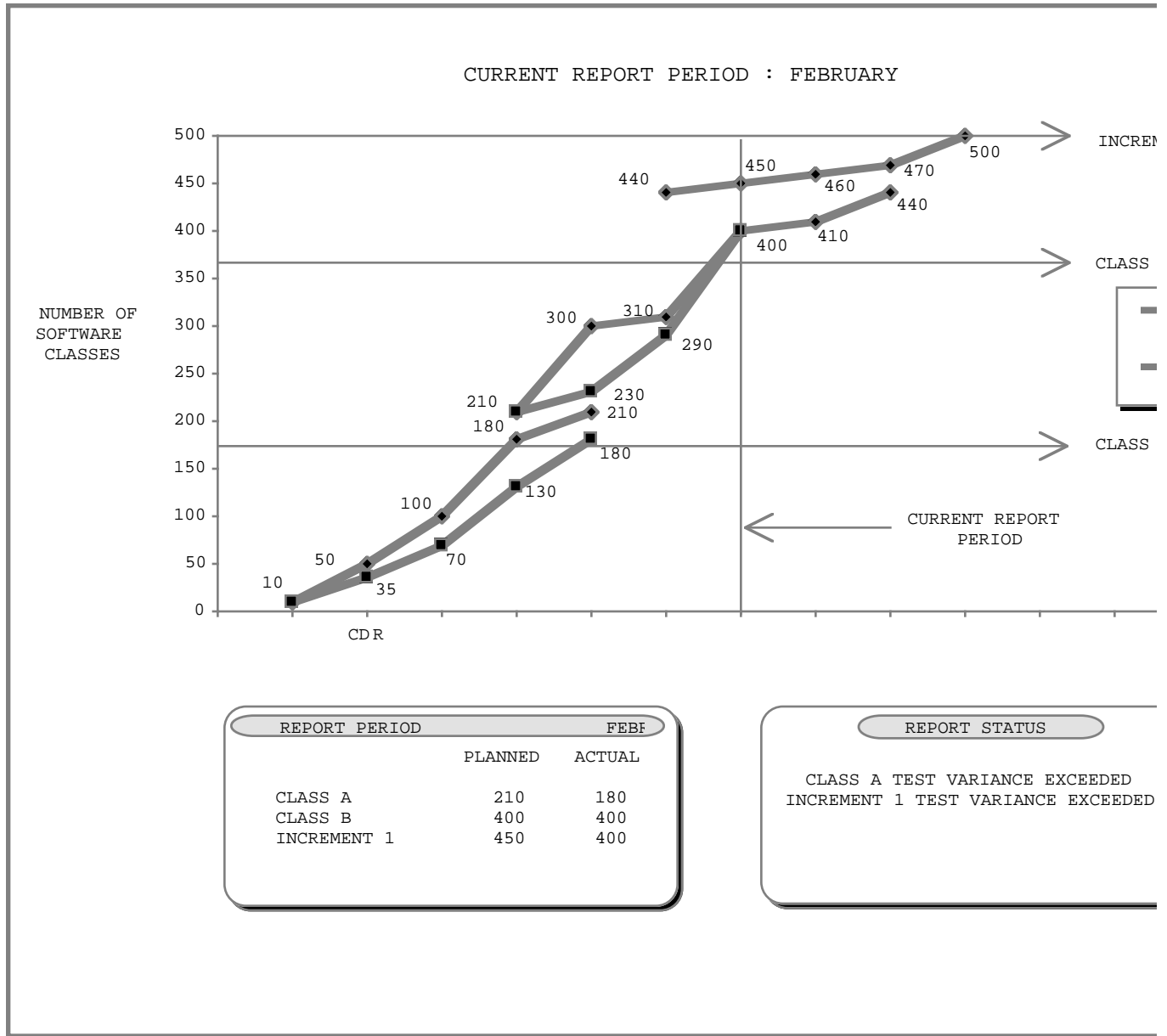


Figure 3.6.2-2 Software Class/Build Progress

**PURPOSE:** Track development organization's ability to test a system based on project requirements. Provide a program manager with an indication of the adequacy of the test program and quality of delivered products. Support analysis of product development processes.

**INPUT DATA:** Number of new Software Problem Reports (SPRs)  
Cumulative number of open SPRs  
SPR density

**TRACKING PERIOD:** CSCI testing (TRR) through Post Deployment Software Support for each planned activity

**FREQUENCY OF COLLECTION:** Monthly

**USAGE:** Defects are anomalies noted during the design, development, and testing process. Software defect data is gathered through SPRs. The SPR density metric provides an indication of the adequacy of the test program and the quality of the tested product. Although too few SPRs may indicate code of exceptional quality, more often it suggests a test program that does not adequately exercise code segments. Too many SPRs may indicate areas where component quality is unacceptable. A "normal" range for SPR density is between 10 and 20 per 1000 SLOC. Figure 3.7.2-1 is an example of a reporting structure providing visibility during integration and test. For the current reporting period, both the number of open SPRs and the SPR density fall within project limits. If the slope of the open SPR graph was positive, it would indicate that problems are being identified faster than they can be resolved. Alternately, if the slope was negative, then it is possible to extrapolate the graph and project a resolution date. Although it is expected that new SPR counts will rise at the start of a testing activity, CSCI SPR counts should approach zero as PCA approaches for any confidence in current schedules.

For more complex systems, it may be necessary to duplicate this graph for selected CSCIs or provide additional graphs based on the classification of the SPR or the priority of the SPR.

For example, the MIL-STD-498 development standard identifies five distinct problem classification categories, based on the priority of the problem description. Separate metrics graphs might be required for each of the categories. For object oriented development, SPR graphs for objects or class structures may be more appropriate.

**TRIGGERING GUIDELINE:** 10 - 20 SPRs per 1000 estimated SLOC

**ACTIONS:** Detailed explanation from the developing organization and related discussions regarding resource or schedule improvements.

**SPECIAL DEFINITIONS:** SPR Density - cumulative number of SPRs per 1000 SLOC

Software Problem Report - A document (electronic or hard copy) used to recognize, record, track, and close anomalies detected in the software and its accompanying documentation.

Defect - A product's inconsistency with its specification. Examples include omissions and imperfections found in the software during definition, design, implementation, and testing.

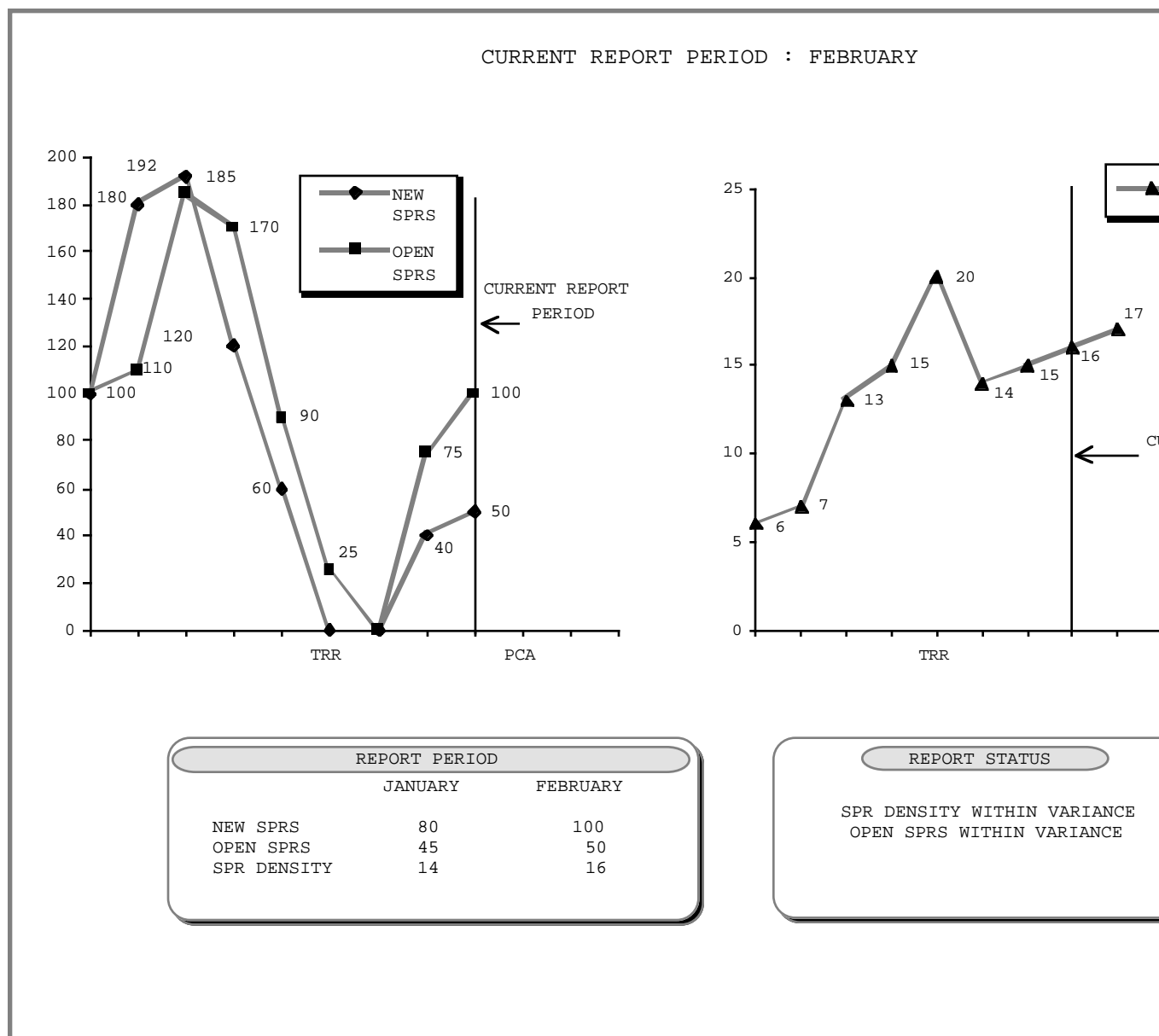


Figure 3.7.2-1 Software Defect Reports

**NOTES:** SPR data formats should be capable of supporting quality-oriented reporting and process-oriented, root causal analysis (e.g., longevity of open SPRs, error injection type, type of software affected, SPR density by category, etc.). Appendix C contains some guidelines for structuring SPR collection forms to support these goals. [FLORAC 92] provides examples of checklists used to construct problem and defect measurement definitions and specifications. Models are provided for the following SPR attributes: Identification, Problem Status, Problem Type, Uniqueness, Criticality, Urgency, Finding Activity, Finding Mode, Date/Time of Occurrence, Problem Status Date, Originator, Environment, Defects Found In, Changes Made To, Related Changes, Projected Availability, Released/Shipped, Applied, Approved By, and Accepted By.

SPR data counts should also be used to support efficiency/effectiveness analysis of the testing program (Section 3.5). Efficiency analysis should include trade-offs on the amount of effort expended to uncover defects and the value of continued testing within selected code segments. Effectiveness measures require continued collection of SPR data during PDSS to calculate test/PDSS defect ratios.

### 3.7.3 Additional Questions

SPRs may be tracked for any product or work item that has passed through its exit criteria and was counted as completed by the development progress measures. Other examples of defects and potential entry/exit criteria include [ROZUM 92]:

- Errors detected during unit-level testing- the work item would be the coding of a unit and the exit criteria could be successful compilation, unit test, and code inspection.
- Specification components- the work item would be the specific CDRL item and the exit criterion is its delivery.
- Action Items from reviews- the work item would be the product(s) being reviewed and the exit criterion is the review.

Based on program specific issues, the COR/Program Manager might also want to consider the following software defect measurements and partitions :

- priority, severity, or criticality of the defect [MIL-STD-498]
- software language
- development process or activity that caused the defect
- development process or activity that found the defect
- effort expended to close defect ( or categories of defects)

Software Defect measurements can be used to help answer the following question :

- Does the quality of the product indicate that the product is ready for release to the customer/user?
- Will undetected or unresolved problems in the product lead to more problems in the next life-cycle stage?
- Does the number of SPRs indicate that the software product should be reworked before proceeding to the next life-cycle stage?

- Is the testing activity complete ( correlate with test progress metric)?
- Are project personnel addressing the trouble reports in a timely manner?
- Do the types of defects suggest areas for process improvement?
- How does this project compare to others with regard to the number and types of defects discovered?
- Which CSCIs tend to be more error prone?
- Is the efficiency at the testing program consistent with defect densities?

## **3.8 Software Effort**

### **3.8.1 Purpose**

Software Effort measurements provide Contracting Office Representative(CORs) and development and post deployment software support personnel with the relationship between planned and actual staff months (hours) expended. Effort measures allow the Program Manager to track the contractor's effort and make inferences about project costs. The Program Manager tracks the number of staff months (hours) expended monthly starting at contract award and compares planned with actual level of expenditures. Planned staffing hours are recorded in the contractor's proposal and development plans.

### **3.8.2 Description**

METRIC CATEGORY: Resources - Cumulative Effort

PURPOSE: Monitors total number of staff-months (hours) expended against the system, CSCI, Software Units, or specified WBS element.

INPUT DATA: Number of staff-months (hours)  
Cumulative number of staff-months (hours)

TRACKING PERIOD: Full Life-cycle for each planned activity

FREQUENCY OF COLLECTION: Monthly

USAGE: Although effort metrics are historically applied to tune costing models and establish productivity values by adding projected staff-month data to monthly reports, discrepancies between planned and actual loading can be analyzed. Figure 3.8.2-1 provides an example graph with planned staff-month loading overlaid.

TRIGGERING GUIDELINE: For graphs with projected loading included, 10% difference between planned and actual loading;

ACTIONS: Detailed explanation from the developing organization regarding discrepancies in project personnel assignment;

SPECIAL DEFINITIONS: Staff-month equates to 153 hours/month;

NOTES: Software staff includes the engineering and management directly responsible for software planning/management, requirements definition, design, coding, test, documentation, configuration management, quality assurance, and maintenance support activities.

### 3.8.3 Additional Questions

To provide better insight into and control over project staff requirements, staff-hours may be partitioned by:

- Development discipline area (software quality assurance, configuration management, test and integration, etc.)
- Development activity (analysis, design, implementation, etc.)
- WBS elements

Total effort expended when graphed is usually in the form of a flattened S-curve. The S-curve reflects an orderly and achievable increase through software requirements analysis and detailed design, a somewhat constant staffing level, peaking during code and unit test, and an orderly decrease through integration and test stages. If there is a significant under-expenditure of planned and actual staff-hours, the contractor may be having problems staffing the contract. Other possible reasons may include [ROZUM 92]:

- Overestimating the software size : The Program Manager should correlate effort measures with software size measures. If the actual size data is tracking below the plan curve, it may indicate that the original sizing estimates were overestimated. True staffing profiles can then be recalculated using SLOC and productivity data.
- Insufficient development progress: By correlating the effort measures with the development progress measures, the Program Manager can determine if under-expenditure of staff hours is causing the development progress measures to track below the plan.
- Reduction in program functionality: The program manager should correlate effort measures with the number of modified or deleted program requirements.
- Increasing levels of open problems: If the difference between the number of open and closed SPRs is increasing, additional staff may be needed or redirected to correct outstanding defects.

Conversely, if there is significant over-expenditure of staff hours, the contractor may have been forced to absorb staff members from other projects or introduce staff members early to capture resources. Other possible reasons for the over-expenditure of staff hours may include:

- Underestimating the size of the software: The Program Manager should correlate effort measures with software size measures.
- Insufficient development progress: By correlating effort measures with the development and milestone performance measures, the Program Manager can determine if the contractor is trying to make up delays by adding staff to the contract.
- Increasing number of defects: By correlating effort measures with the software defects measures, the Program Manager can determine if the contractor is adding staff to correct a growing number of SPRs.
- Growth in program functionality: The program manager should correlate effort measures with number of new, modified program requirements.
- Inability to capitalize on reuse: The program manager should correlate effort measures with the migration of reused code segments to newly developed code categories and the subsequent increase in staffing overhead.

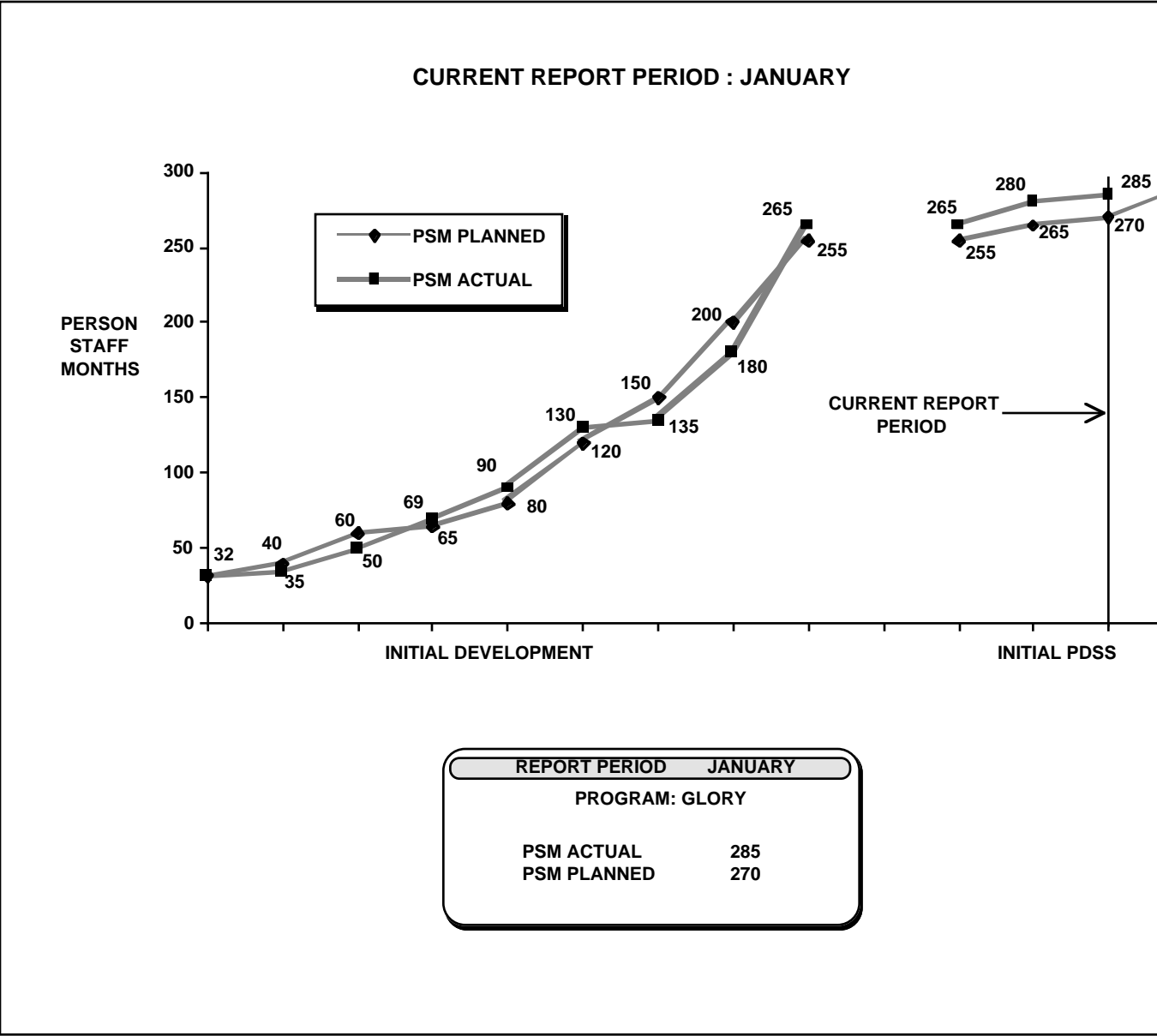


Figure 3.8.2-1 Cumulative Effort

Based on differences between planned and actual effort data, it may be beneficial to collect personnel specific measures. One example measure is the number of personnel expected, number of personnel actual, and staff turnover. Staff turnover data may reveal potential schedule and performance problems despite reasonable total effort profiles.

### **3.9 Software Problem Resolution Effort**

#### **3.9.1 Purpose**

Software Problem Resolution Effort measurements provide Contracting Office Representative (CORs) and development and post deployment software support personnel with an indicator of the average effort required to resolve SPRs. Although average resolution time increases as testing proceeds from the Unit-level to the CSCI integration and test stages, unbounded increases suggest problems with Unit-level test processes or the definition / use of software interfaces.

#### **3.9.2 Description**

METRIC CATEGORY: Quality - Problem Resolution Effort

PURPOSE: Monitors cumulative average effort required to resolve software problem reports;

INPUT DATA: Number of resolved software problem reports  
Time in hours to resolve related problems

TRACKING PERIOD: Software Integration and Test through Post Deployment Support for each planned activity

FREQUENCY OF COLLECTION: Monthly

USAGE: As the Software Units/CSCIs are integrated, test personnel document errors encountered while executing formal test procedures through software problem reports. An example software problem report is given in [HAGER 89]. During the integration testing, test procedures exercise the interfaces to an increasing number of software elements. Modifications required at this level will require more engineering time to resolve than previously performed Unit-level testing. However, it is expected that the slope of the resolution time should “flatten” and stabilize. Constantly increasing positive slopes, as shown in Figure 3.9.2-1, suggest problems with interface definition and implementation. Figure 3.9.2-2 and Figure 3.9.2-3 show alternative formats for displaying SPR reports.

TRIGGERING GUIDELINE: None

ACTIONS: N/A

SPECIAL DEFINITIONS: None

NOTES: To provide more visibility to the processes contributing to resolution times, it may be necessary to graph resolution effort data by work category (e.g., redesign, Unit-level testing, CSCI Integration and Test stage, System Test stage, etc.).

### 3.9.3 Additional Questions

Software Problem Resolution Effort measures provide an indicator of a contractor's ability to adequately/successfully test a system. Problem Resolution Effort measurements can be used to help answer the following questions.

- Are test plans/procedures adequate to exercise program segments?  
Are specific CSCIs/Software Units generating defects requiring significantly more resolution time, thus requiring more program visibility (schedules, specifications, test structures, etc.)?
- Does the SPR average resolution time suggest problems with definition or use of specific interfaces?
- Does the SPR average resolution time indicate schedule slippage and the necessity of a replanning effort at the program level (schedule and resources)?

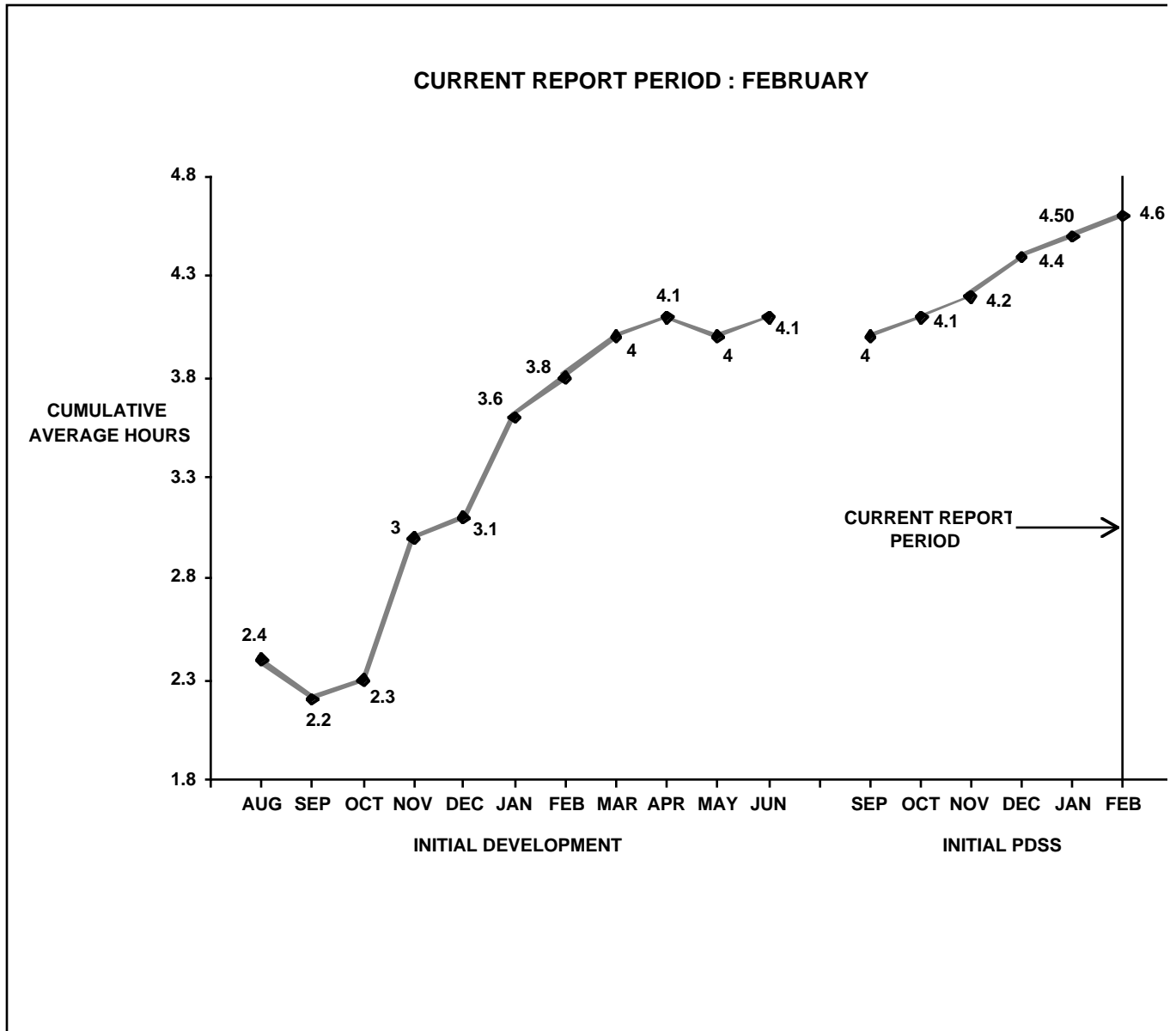


Figure 3.9.2-1 Cumulative Average SPR Resolution Effort

## CURRENT REPORT PERIOD

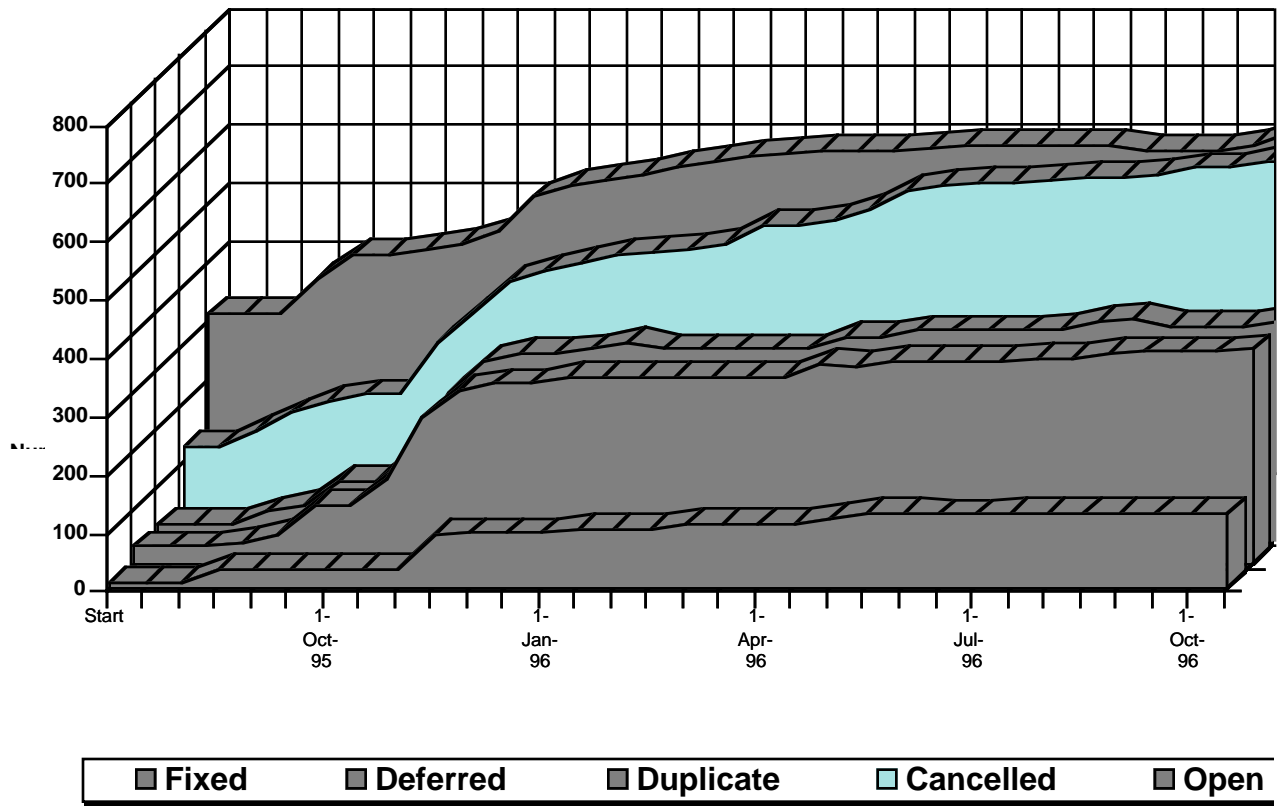


Figure 3.9.2-2 Closed SPRs

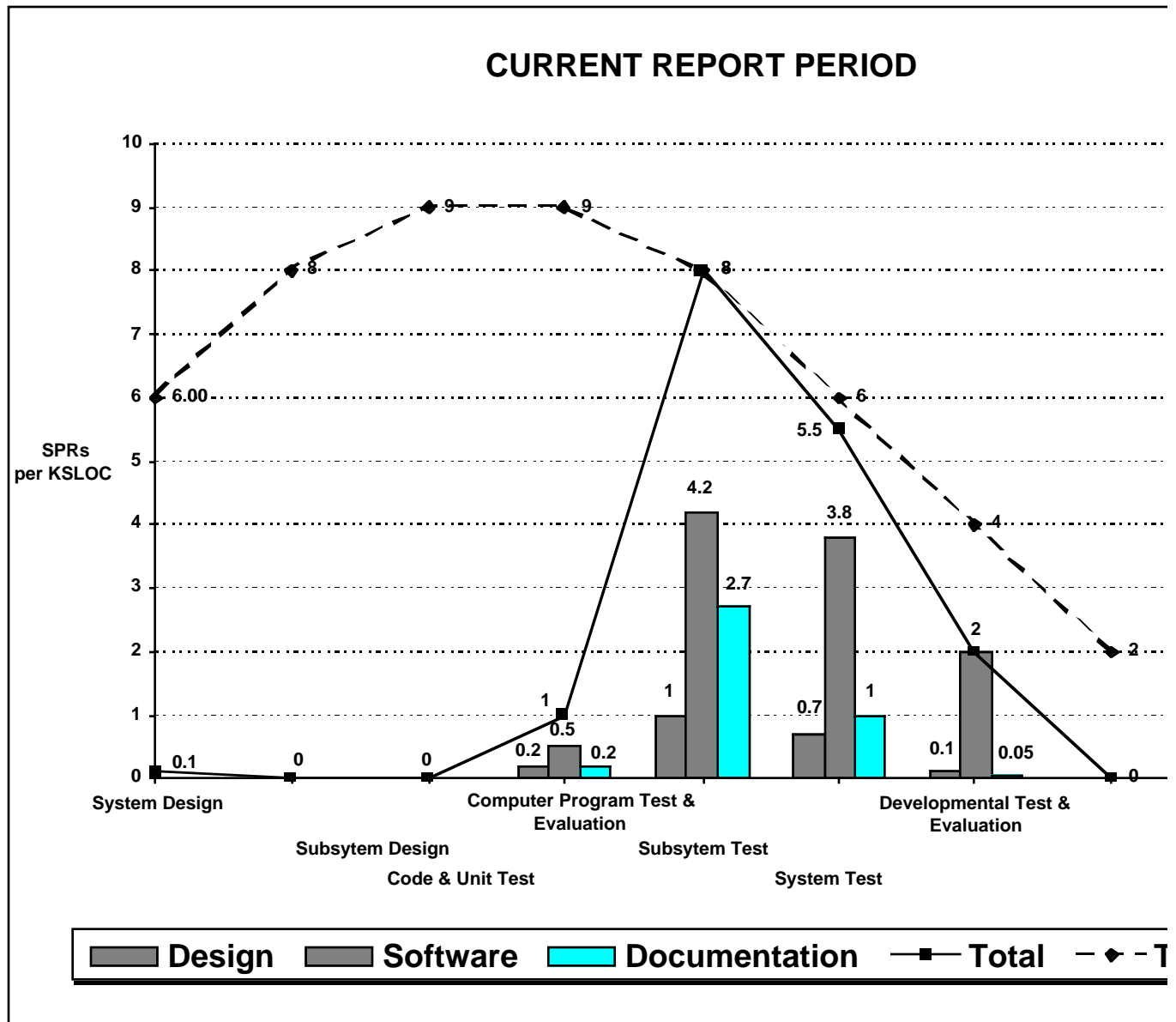


Figure 3.9.2-3 Valid SPRs By Phase Found and Category

#### 4. REFERENCES

---

- [AIRF 86] Department of the Air Force, Air Force Systems Command Software Management Indicators (AFSC Pamphlet 800-43). Washington, DC, Andrews Air Force Base, 1986.
- [BASILI 84] Basili, V. R., and Weiss, D. M. "A methodology for collecting valid software engineering data." IEEE Trans. Software Eng. SE-10 (6 1984): pp. 728-738.
- [BAUMERT 92] Baumert, J.H., and McWhinney, M.S., Software Measures and the Capability Maturity Model. Software Engineering Institute, Technical Report, CMU/SEI-92-TR-25.
- [BETZ 91] Betz, H., and O'Neill, P., "Army Software Test and Evaluation Panel (STEP) Software Metrics Initiatives Report (Draft)." March 1991.
- [BOEHM 81] Boehm, B., Software Engineering Economics. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
- [CHIDAMBER 94] Chidamber, S.R. and Kemerer, C. F., "A Metrics Suite for Object Oriented Design", IEEE Trans. on Software Engineering, vol. 20, No. 6, pp. 476-93, June 1994.
- [CHMURA 90] Chmura, L. J., Norcio, A. J., and Wicinski, T. J. "Evaluating Software Design Processes by Analyzing Change Data Over Time." IEEE Trans. on Soft. Eng. 16, 7, pp. 729-740, 1990.
- [CONSORTIUM 94] Software Measurement Guidebook, SPC-91060-CMC, August 1994.
- [DECKER 91] Decker, W., Baumert, J., Card, D., Wheeler, J., and Wood, R., SEAS Software Measurement System Handbook. (CSR/TR-89/6166), Beltsville, MD, Computer Sciences Corporation, 1991.
- [DEMARCO 82] DeMarco, T., Controlling Software Projects. New York : Yourdon Press, 1982.
- [DOD2167A 88] DoD-STD-2167A, Military Standard, Defense System Software Development, 29 February, 1988.
- [FENTON 91] Fenton, N.E., Software Metrics: A Rigorous Approach, Chapman and Hall, New York, New York, 1991.
- [FLORAC 92] Florac, W., Software Quality Measurement: A Framework for Counting Problems and Defects. Software Engineering Institute, Technical Report, CMU/SEI-92-TR-22.
- [GOODMAN 93] Goodman, Paul, Practical Implementation of Software Metrics. McGraw-Hill, Inc., 1993.
- [GOETHERT 92] Goethert, W. B., Bailey, E. K., and Busby, M. B., Software Effort and Schedule Measurement: A Framework for Counting Staff-Hours and

- Reporting Schedule Information. Software Engineering Institute, Technical Report, CMU/SEI-92-TR-21.
- [GRADY 87] Grady, R. and Caswell, D., Software Metrics: Establishing a Company-Wide Program. Englewood Cliffs: Prentice-Hall, Inc., 1987.
- [GRADY 92] Grady, R., Practical software metrics for project management and process improvement. Englewood Cliffs: Prentice Hall, Inc., 1992.
- [HAGER 89] Hager, J.A., "Software Cost Reduction Methods In Practice: A Post-Mortem Analysis." IEEE Trans. on Soft. Eng., December 1989.
- [HIHN 91] Hihn, J., and Habib-agahi, H. "Cost Estimation of Software Intensive Projects: A Survey of Current Practices," 13th International Conference on Software Engineering, Austin, Tx, USA, May 1991.
- [HUMPHREY 89] Humphrey, W., Managing the Software Process. Reading, MA., Addison-Wesley Publishing Company, 1989.
- [HUMPHREY 95] Humphrey, W., A Discipline for Software Engineering. Reading, MA., Addison-Wesley Publishing Company, 1995.
- [IEEE 92] Standard for Software Productivity Metrics. Washington, D.C., The Institute of Electrical and Electronics Engineers.
- [JONES 94] Jones, Capers. "Software Metrics: Good, Bad, and Missing" , Computer, vol. 27, pp. 98-100, September 1994.
- [JONES 95] Jones, Capers.
- [KIM 93] Kim, E. M., *An experimental evaluation of OOP complexity metric: SOMEFOOT*, Master thesis, Chonbuk National University (1993).
- [KIM 94] Kim, E.M. and Chang, O.B. and Kusumoto, S. and Kikuno, T. "Analysis of Metrics for Object-Oriented Program Complexity", Proceedings - IEEE Computer Society's International Computer Software & Applications Conference, p 201-207, 1994.
- [LANDIS 90] Landis, L., McGarry, F., Waligora, S., Pajerski, R., and Start, M., Manager's Handbook for Software Development - Revision 1. (SEL-84-101), Greenbelt, MD, NASA Goddard Space Flight Center, 1990.
- [LARANJEIRA 90] Lanranjeira, Luiz A. "Software Size Estimation of Object-Oriented Systems." IEEE Transactions on Software Engineering, Vol. 16, No. 5, pp. 510-522, May 1990.
- [LI 93] Li, W. and Henry, S. "Object-Oriented metrics that predict maintainability," The Journal of Systems and Software, vol. 23, pp. 111-122, 1993.
- [LORENZ 93] Lorenz, M., Object-Oriented Software Development: A Practical Guide, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

- [LORENZ 94] M. Lorenz, J. Kidd, Object-Oriented Software Metrics, Prentice Hall, Englewood Cliffs; 1994.
- [MATSON 94] Matson, Jack E. and Barrett, Bruce E., and Mellichamp, Joseph M., "Software Development Cost Estimation Using Function Points." IEEE Trans. on Soft Eng., Vol. 20, No. 4, April 1994.
- [MCCABE 89] McCabe, T.J., and Butler, C.W., "Design Complexity Measurement and Testing." Communications of the ACM, December 1989.
- [MCCABE 95] McCabe's brochures 1995.
- [MURINE 83] Murine, G. E., "Improving Management Visibility Through the Use of Software Quality Metrics." Proceedings from IEEE Computer Society's Seventh International Computer Software and Application Conference, New York, N.Y., 1983.
- [NSAM 81] NSA/CSS Software Product Standards Manual. NSAM 81-3/DoD-STD-1703, April 1987.
- [PAULK 91] Paulk, M., Curtis, W., and Chrissis, M., Capability Maturity Model For Software. Software Engineering Institute, Technical Report, CMU/SEI-91-TR-24.
- [ROZUM 92] Rozum, J. A., Software Measurement Concepts for Acquisition Program Managers. Software Engineering Institute, Technical Report, CMU/SEI-92-TR-11.
- [SCHULTZ 88] Schultz, H.P., Software Management Metrics. MITRE Corporation, May 1988.
- [SHEPPERD 93] Shepperd, Martin, Software Engineering Metrics Volume 1: Measures and Validations. McGraw-Hill, Inc., 1993.
- [SHEPPERD 94] Shepperd, Martin, and Ince, D.C., "A Critique of Three Metrics," Journal of Systems Software, 26, pp.197-210, 1994.
- [SYMONS 91] Symons, Charles R., Software Sizing and Estimating: Mk II FPA (Function Point Analysis). John Wiley & Sons. 1993.
- [VERNER 92] Verner, June and Tate, Graham "A Software Size Model." IEEE Trans. on Soft. Eng., Vol. 18, No. 4, April 1992.
- [YEH 93] Yeh, Hsiang-Tao, Software Quality Process. Mc Graw-Hill, Inc., 1993.
- [WALSH 79] Walsh, T.J., "Software Reliability Study Using A Complexity Measure." Proceedings of the National Computer Conference, New York: AFIPS, 1979.
- [WILLIAM 93] Williams, John D. "Metrics for Object Oriented Projects" OOPSLA '93 Workshops Program: Process and Metrics for Object-Oriented Software Development , Vancouver, Canada, 1993.

[ZUSE 91]

Zuse, Horst. Software Complexity: Measures and Methods., Walter de Gruyter, 1991.

## APPENDIX A : ACRONYMS

---

<u>AC</u>	Actual Complexity
<u>APD</u>	Access to Public Data
<u>BCWP</u>	Budgeted Cost Of Work Performed
<u>BCWS</u>	Budgeted Cost Of Work Scheduled
<u>CC</u>	Cyclomatic Complexity
<u>CDR</u>	Critical Design Review
<u>CDRL</u>	Contract Data Requirements List
<u>CI</u>	Code Inspection
<u>COR</u>	Contracting Office Representative
<u>COTS</u>	Commercial Off-the-Shelf
<u>CSCI</u>	Computer Software Configuration Item
<u>DC</u>	Design Complexity
<u>DI</u>	Design Inspection
<u>DOD</u>	Department of Defense
<u>DRR</u>	Design Ready Review
<u>FCA</u>	Functional Configuration Audit
<u>FQT</u>	Formal Qualification Test
<u>HW</u>	Hardware
<u>IC</u>	Integration Complexity
<u>IDD</u>	Interface Design Document
<u>IEEE</u>	Institute of Electrical and Electronics Engineers
<u>IRS</u>	Interface Requirements Specification
<u>IV&amp;V</u>	Independent Verification and Validation
<u>LOCM</u>	Lack of Cohesion in Methods
<u>MDC</u>	Module Design Complexity
<u>NOC</u>	Number of Children
<u>OT&amp;E</u>	Operational Test and Evaluation
<u>O-O</u>	Object-Oriented
<u>PCA</u>	Physical Configuration Audit
<u>PDL</u>	Program Design Language
<u>PDR</u>	Preliminary Design Review
<u>PM</u>	Project Manager
<u>PPPD</u>	Percentage of Public and Protected Data
<u>QA</u>	Quality Assurance
<u>RFP</u>	Request for Proposals
<u>SDD</u>	Software Design Document
<u>SDR</u>	System Design Review
<u>SEI</u>	Software Engineering Institute
<u>SLOC</u>	Source Line Of Code
<u>SOW</u>	Statement Of Work

<u>SPR</u>	Software Problem Report, or alternatively, Special Procurement Request
<u>SRS</u>	Software Requirements Specification
<u>TRR</u>	Test Readiness Review
<u>WBS</u>	Work Breakdown Structure

## APPENDIX B : TERMS

---

Anomaly - Anything observed in the documentation or operation of software that deviates from expectations based on previously verified software products or reference documents.

Baseline - A set of verifiable products placed under configuration management, that represent a benchmark in the system development process.

Budgeted Cost of Work Scheduled - cost estimates for individual work packages as determined by engineering inputs and grouped via a Work Breakdown Structure.

Budgeted Cost of Work Performed - cost actuals for individual work packages completed as determined by monthly cost accounting and the Work Breakdown Structure

Causal Analysis - The analysis of defects to determine their underlying root cause.

Computer Software Configuration Item (CSCI) - A configuration item for software [MIL-STD-498].

Software Unit - A separately, testable element, specified in the design of a computer software component [MIL-STD-498].

Code Inspection - A review by peers and management to examine code for conformance with program standards, satisfaction of the intended design, detection of logic errors or misinterpretation of the design specification.

Cyclomatic Complexity - The number of unique logical paths in a software element. Cyclomatic complexity values correlate directly with the number of required test cases needed to verify the logical paths.

Defect - (1) Any unintended characteristic that impairs the utility or worth of an item, (2) Any kind of shortcoming, imperfection or deficiency, (3) Any flaw or imperfection in the software work product or software process.. Examples include omissions and imperfections found in software during definition, design, implementation, and testing.

Design Inspection - A formal review by peers and management to examine software design for conformance to program standards, satisfaction of allocated requirements, detection of design errors or misinterpretation of requirements, assessment of design simplicity, modularity and testability, and determining adequacy of test plans.

Design Ready Review - a formal review by peers and management to determine if a designer has all the information necessary to complete a detailed design, and understands the Unit requirements and file/inter- Unit interfaces.

Failure - The inability of a system or component to perform its required functions within the specified performance requirements.

Formal Review - A formal meeting at which a product is presented to the end-user, a customer, or other interested parties for comment and approval [MIL-STD-498].

Integration Ready Review - A formal review of an implemented and tested Unit for the purpose of determining if the unit is ready to be turned over to the integration and test function.

Phase (Stage) - An increment of the system development life-cycle which has clearly defined inputs, activities, and products as described by the engineering methodology.

Program Design Language (PDL) - A design tool used at any required level of detail to facilitate the translation of functional specifications into computer instructions.

Software Problem Report - A document (electronic or hard copy) used to recognize, record, track, and close anomalies detected in the software and its accompanying documentation.

Software Problem Report Density - cumulative number of software problem reports normalized by program-defined blocks of source code ( e.g., SPRs per 1000 source lines of code).

Source Line Of Code - Instructions created by project personnel and translated into machine code; includes job control language, data declarations, and format statements; it excludes comment statements [IEEE 92].

Unit - A software work package to which the satisfaction of requirements can be traced and which can be designed and developed by one engineer.

Work Breakdown Structure - a mechanism for organizing the program work to be accomplished into a hierarchical structure and for defining discrete tasks to be performed in fulfilling the technical and management requirements of the program.

Staff Hour - An hour of time expended by a member of the program staff

Staff Month - Number of staff hours per month as defined in program profile checklists

## APPENDIX C : SOFTWARE DEFECT REPORT GUIDELINES

---

### 1 Introduction

A key component of any software measurement program targeted at establishing and maintaining control over the development and maintenance of a software product is the collection and analysis of software problem and defect data. Software defect measurements have direct application to: estimating, planning, and tracking the various software development processes, to determining the status of corrective action, to measuring and improving the software development process, and to the extent possible, predicting remaining defects or failure rates [MURINE 83]. By measuring problems and defects, program managers obtain the data necessary to control such program attributes as:

**Quality:** The number and frequency of problems and defects associated with a software product are inversely proportional to the quality of the software. Software problems and defects are among the few direct measurements of software process and product quality. These measurements allow a program manager to quantitatively describe trends in defect or problem discovery, repairs, process and product imperfections.

**Cost:** Rework is a significant cost factor in software development and maintenance. The number of problems and defects associated with the product directly contributes to this cost. Measurement of the problems and defects helps a program manager understand where and how the problems and defects occur and provides insight to the methods of detection, prevention, and prediction.

**Schedule:** Program managers can use problem and defect measurements in tracking project progress, identifying process inefficiencies, and forecasting obstacles that will jeopardize schedule commitments.

### 2 Problem and Defect Reporting

To establish a software measurement environment, the software organization must define a data collection process and recording media. Typically, Software Problem Reports (SPR) are the mechanism used to collect data about problems and defects. Configuration Management report [LONDON 95] is a typical example of SPR reporting requirements and procedures. SPRs give rise to additional measurement communication issues. Problem reports generated by “finding activities” (e.g., inspections, formal reviews, testing, etc.) are typically tuned to the needs of the life-cycle activity and vary in content and format. The problems are recorded and reported at different points in time, in batches or continuously, by different organizations (developer and customer), by people with varying degrees of understanding of the software product. Often the data is recorded in separate databases or captured with non-compatible record-keeping mechanisms and with very little concern for root-causal analysis requirements. A common format and language is required to bridge these differences and provide a consistent basis for communicating. Problem report checklists [FLORAC 92] are required to record problem and defect recording, reporting, and definition assumptions.

### 3 Software Problem Report Attributes

Despite the variances in the way software problems are recorded and reported, there are many similarities among the reports. This section provides some guidelines for the type of information included on SPR forms [FLORAC 92]. Program-specific assumptions should be recorded via checklists and recorded in standard program planning documentation. The following attributes

provide a basis for communicating, descriptively and prescriptively, the meaning of problem and defect measurements.

<u>Attribute</u>	<u>Question Answered</u>
Identification	What software product is involved?
Finding Activity	What process discovered the problem or defect?
Finding Mode	How was the problem or defect found?
Criticality	How critical or severe is the problem or defect?
Problem Status	What work needs to be done to dispose of the problem?
Problem Type	What is the nature of the problem ? If a defect, what kind?
Uniqueness	What is the similarity of this problem to previous problems or defects?
Urgency	What priority is associated with the problem?
Environment	Where was the problem discovered?
Timing	When was the problem discovered? When was it reported? When was it corrected?
Originator	Who reported the problem?
Defects Found In	What software artifacts caused or contained the defect?
Changes Made To	What software artifacts were changed to correct the defect?
Projected Availability	When are changes expected?
Released/Shipped	What configuration level contains the changes?
Applied	When was the change made to the baseline configuration?
Approved By	Who approved the resolution of the problem?
Accepted By	Who accepted the problem resolution?

#### 4 Attribute Descriptions

**Problem ID:** This attribute serves to uniquely identify each problem for reference purposes.

**Product ID:** This attribute identifies the software product to which the problem refers. It should include the Version and Release ID for released products, or the Build ID for products under development.

**Problem Status:** This attribute refers to a point in the problem analysis and corrective action process where some program-specified progress criteria have been met. The problem status is of interest to the program manager because it reveals information about the development organization's ability to resolve and dispose of the reported problems. The recognition or opening of a problem is based on the existence of data describing the event. As the investigative work proceeds, more data is collected about the problem, including the information that is required to satisfy the issues raised by the problem. It is the existence of this data that is used as a set of criteria to satisfy moving from status OPEN to status CLOSED. Typical Problem Status attributes might include:

Open: The problem is recognized and some level of investigation and action will be undertaken to resolve it.

Recognized: Validity of problem report data has been established and sufficient data has been collected to permit an evaluation of the problem to be made.

Evaluated: Sufficient data has been collected by investigation of the reported problem and the various software artifacts to at least determine the problem type. Depending on the software organization, the amount of data required to satisfy the criteria for this state may vary significantly.

Resolved: The problem has been reported and evaluated, and sufficient information is available to satisfy the rules for resolution. Each organization will have its own set of processes or activities related to resolution (described in program planning documentation). These processes may include the proposed change, change control approval, and a root causal analysis of the problem.

Closed: The investigation is complete and the action required to resolve the problem has been proposed, accepted, and completed to the satisfaction of all involved. In some cases, the problem report will be recognized as invalid as part of the recognition process and closed immediately.

**Problem Type:** This attribute is used to assign a value to the problem that will facilitate the evaluation and resolution of the reported problems. It is used to classify the problems into one of several categories to support problem resolution and ultimately root-causal analysis. Examples of Problem Type fields include:

Requirements Defect: A mistake made in the definition or specification of the customer needs for a software product. This includes defects found in functional specifications; interface, design, and test requirements; and program deliverable requirements specifications.

Design Defect: A mistake made in the design of a software product. This includes defects found in functional descriptions, interfaces, control logic, data structures, error handling, conformance with design standards, and program deliverable design specification.

Code Defect: A mistake made in the translation of design specifications to code segments. This includes defects found in program logic, interface handling, data definitions, computation, adherence to program specified coding standards, and program deliverable code specifications.

Document Defect: A mistake made in a software product publication. This does not include mistakes made to requirements, design, or coding documents.

Test Case Defect: A mistake made during procedural definition or execution of test structures.

Examples of non-software oriented Problem Type fields include:

Hardware Problem: A problem due to a hardware malfunction that the software does not, or cannot, provide fault tolerant support.

Operating System Problem: A problem that the operating system(s) in use has responsibility for creating or managing.

User Error: A problem due to user misunderstanding or incorrect use of the software.

Operations Error: A problem caused by an error made by the computer system operational staff.

**New Requirement/Enhancement:** A problem that describes a new requirement or functional enhancement that is outside the scope of the software product baseline requirements.

**Undetermined Problem:** Information provided with the problem is not sufficient to assign a problem type.

**Uniqueness:** This attribute differentiates between a unique problem or defect and a duplicate. Possible values are:

**Duplicate:** The problem or defect has been previously discovered.

**Original:** The problem or defect has not been previously reported or discovered.

**Value Not Identified:** An evaluation has not been made.

**Priority:** Provides a measure of the impact a problem has on developed software or customer mission. Criticality is usually measured with several levels, the most critical being mission or life-threatening, and the least being a minor operational impact [MIL-STD-498]. The assigned priority determines the order in which problems are evaluated, resolved, and closed.

**Finding Activity:** Refers to the activity, process, or operation taking place when the problem was encountered. Instead of using the program development phases or stages to describe the activity, specific development activities should be used. An example of a Finding Activity hierarchy checklist is:

Finding Activity

Synthesis of:

- Requirements
- Design
- Code
- Test Procedures
- User Publications

Inspections of:

- Requirements
- Software Architecture
- Unit Detailed Design
- Operational Documentation
- Test Procedures

Formal Review of:

- Program Plans
- Requirements
- Preliminary Design
- Detailed Design
- Test Readiness
- Formal Qualification

Testing

- Planning
- Unit-level
- CSCI-level
- Subsystem Integration and Test
- System Integration and Test
- Customer Acceptance

Customer Support  
Deployment  
Installation  
Operation

**Finding Mode:** This attribute is used to identify whether the problem was discovered in an operational environment or in a non-operational environment. The values for this attribute are: Dynamic or Static. Dynamic attributes describe problems or defects found during the operation or execution of the computer program. Static attributes describe problems found in a non-operational mode. Examples include problems found during formal reviews, peer inspections, or other activities that do not require execution of the software.

**Date/Time of Occurrence:** The date/time data is useful in establishing and predicting software reliability, failure rates, and numerous other time-related measurements. It is also necessary to recreate problems that are date or time-of-day dependent.

**Problem Status Dates:** These attributes refer to the date on which the problem report was received or logged. The data is used to determine status, problem age, and problem arrival rate.

**Originator:** The originator attribute provides the information needed by the problem analyst to determine the originating person, organization, or site. This data is useful in determining if a particular problem is location sensitive or in eliminating a problem based on the peculiarities of the operating environment. Additionally, the originator field could be used as a discriminator for aggregating problem report counts.

**Environment:** This attribute provides information needed by the analyst to determine if a problem is uniquely related to an operating environment. This information is important if the analyst is to accurately recreate the problem. This data is also useful in determining if a particular operating environment contributes significantly more reported problems.

**Defects Found In:** This attribute records the software CSCI and/or the Units containing defects causing a problem. This information is useful when determining the reliability of a particular software element or the reliability of a software configuration level following a new release.

**Changes Made To:** This attribute identifies the software Unit(s) changed to resolve the discovered problem or defect.

**Projected Availability:** Identifies the Date when the product fix is committed to be available and the particular Release/Build designator.

**Released / Shipped:** This field contains the date the product fix is released and the Release or Build ID in which the product fix is included.

**Approved By:** Signatures indicating that the product modifications have been approved by appropriate project management.

**Accepted By:** Signatures indicating that the product modifications has been accepted by the appropriate project management.

THIS PAGE INTENTIONALLY LEFT BLANK

## APPENDIX D : OBJECT-ORIENTED METRICS

---

Appendix D contains additional examples of potentially useful object-oriented measurements. Although many of the measures show promise, their near term use in development programs is limited until the empirical data necessary to demonstrate their value is collected.

Table D-1 provides a summary of the object-oriented categories and the associated measures. Immediately following is a list of definitions for the terms used within the table. Detailed descriptions of the measures, recommended triggering levels, and potential usage and interpretation are found in [LORENZ 93], [LORENZ 94], and [CHIDAMBER 94].

**TABLE D-1 OBJECT ORIENTED METRICS**

CATEGORY	METRIC
Application Size	Number of scenario Number of keys Number of support classes Average number of support classes
Staffing Size	Person-days per class Average person-days per class Classes per developer
Scheduling	Number of major iterations Number of contracts completed
Method Size	Number of message sends Source lines of code Average method size
Method Internals	Method complexity Strings of message sends Number of parameters
Class Size	Number of public instance methods Number of instance methods Average number of instance methods per class Number of class methods Number of class variables
Class Inheritance	Class hierarchy nesting level Multiple inheritance nesting level
Method Inheritance	Number of methods overridden by a subclass Number of methods inherited by a subclass Number of methods added by a subclass Number of class extension through specialization
Class Externals	Class coupling Number of times a class is reused Number of classes/methods thrown away Number of class-to-class relationships

**TABLE D-1 OBJECT ORIENTED METRICS (Continued)**

CATEGORY	METRIC
Class Internals	Class cohesion Global usage Instance variable usage Average number parameters per method Percentage of functionally oriented code Average number of comment lines per class/method Average number of commented methods Number of problem reports per class or contract

### **SPECIAL DEFINITIONS**

- Class - a template that defines the structure and capabilities of an object instance
- Class Hierarchy - a tree structure that organizes class inheritance
- Class Hierarchy Nesting - the number of subclassing levels from the top in the class hierarchy
- Class Variables - localized globals that provide common objects to all the instances of a class
- Cohesion - measure of the internal binding within a class or object
- Contract - abstraction of a group of related public responsibilities provided by subsystems and classes to their clients
- Coupling - measure of the external binding between classes or objects
- Instance Variables - name that allows one object or instance to refer to another one
- Key Classes - class central to the application domain being automated
- Message Sends - communication between objects via messages; request of service from another object through message sends
- Method - a class behavior or service that is executed when an object receives a message
- Method Complexity - a measure of the amount of work and number of decisions being made by a method
- Method Override - creating a method in a class with the same name as a method in one of its superclasses
- Method Size - a measure of the volume of a method based on attributes such as number of message sends and types of message sends
- Scenario Scripts - sequence of steps made by the user and system to accomplish a certain task
- Specialization - an extension of the behavior of a type of object
- Support Classes - classes that provide basic services or interfaces capabilities to the key classes
- Private methods - method that exists to perform a class function but is not available to other classes
- Public Instance Methods in a Class - methods that are available as services to other classes
- Public Methods - methods that is readily available to other classes and are grouped into contracts

## INDEX

---

## —A—

- accepted by, 69
- access to public data, 28
- acknowledgments, 6
- acronyms, 61
- Actions, 34
  - functionally-oriented complexity, 25
  - object-oriented complexity metrics, 28
  - software builds, 42
  - software defect reporting, 46
  - software development, 34
  - software effort, 49
  - software problem resolution effort, 52
  - software schedule, 32
  - software size, 17
  - software testing, 41
- actual
  - complexity, 24, 25
  - cost of work performed, 32
- Additional Questions
  - object-oriented complexity metrics, 29
  - software builds, 44
  - software defect reporting, 48
  - software development, 37
  - software effort, 50
  - software problem resolution effort, 53
  - software schedule, 32
  - software testing, 41
- applicability, 4
- approved by, 69
- aspects of program development, 11
- Average CSCI Complexity
  - Figure 3.2.2-2, 23
- Average Software Unit Complexity
  - Figure 3.2.2-1, 22

## —B—

- background, 7
- benefits of measurement programs, 7
- budgeted cost of work scheduled, 32

## —C—

- changes made to, 69
- class, 28
- class hierarchy nesting level, 28
- Closed SPRs
  - Figure 3.9.2-2, 55
- code defect, 67
- common guidelines, 9
- computer software configuration Item, 4
- cost, 65
- critical design review, 4
- CSCI cyclomatic complexity, 25
- Cumulative Average SPR Resolution Effort
  - Figure 3.9.2-1, 54

- cumulative effort
  - Figure 3.8.2-1, 51
- cyclomatic complexity, 21, 25

## —D—

- date/time of occurrence, 69
- defect, 46
- defects found in, 69
- Description
  - software builds, 42
  - software defect reporting, 44
  - software effort, 49
  - software problem resolution effort, 52
  - software size, 14
  - software testing, 38
- design
  - complexity, 24, 25
  - defect, 67
- development, 3
- document defect, 67

## —E—

- effective measurement program, 7
- efficiency, 48
- effort, 2
- encapsulation, 26, 27, 28
- environment, 69
- estimated schedule, 32
- external
  - inputs, 15
  - inquiries, 15
  - interfaces, 15
  - outputs, 15

## —F—

- fan in, 27, 28
- Figure 1.1-1 Metrics/Phase Coverage, 5
- Figure 3.1.2-1 Software Size, 16
- Figure 3.1.2-2 Function Point Monthly Report, 18
- Figure 3.2.2-1 Average Software Unit Complexity, 22
- Figure 3.2.2-2 Average CSCI Complexity, 23
- Figure 3.3.2-1 Software Schedule, 31
- Figure 3.4.2-1 Software Development Progress, 35
- Figure 3.4.2-2 Software Development Progress (Object-Oriented), 36
- Figure 3.5.2-1 Software Test Progress, 40
- Figure 3.6.2-1 Software Build Progress, 43
- Figure 3.6.2-2 Software Class/Build Progress, 45
- Figure 3.7.2-1 Software Defect Reports, 47
- Figure 3.8.2-1 Cumulative Effort, 51
- Figure 3.9.2-1 Cumulative Average SPR Resolution Effort, 54
- Figure 3.9.2-2 Closed SPRs, 55
- Figure 3.9.2-3 Valid SPRs By Phase Found and Category, 56
- Figure 3-1 Metric Template and Field Descriptions, 12

Figure 3-2 Generic Life-Cycle Metrics Report Format, 13  
finding

- activity, 68
- mode, 69

#### Frequency of Collection

- functionally-oriented complexity, 21
- object-oriented complexity metrics, 27
- software builds, 42
- software defect reporting, 46
- software development, 33
- software effort, 49
- software problem resolution effort, 52
- software schedule, 30
- software size, 14
- software testing, 39

function points, 15

- Figure 3.1.2-2, 18

functionally - oriented complexity metrics, 20

### —G—

Generic Life-Cycle Metrics Report Format

Figure 3-2, 13

global data complexity, 24, 25

### —I—

inheritance, 26, 27, 28

#### Input Data

- functionally-oriented complexity, 20
- object-oriented complexity metrics, 26
- software builds, 42
- software defect reporting, 46
- software development, 33
- software effort, 49
- software problem resolution effort, 52
- software schedule, 30
- software size, 14
- software testing, 38

integration complexity, 24, 25

internal files, 17

### —L—

lack of cohesion in methods, 28

### —M—

#### Measurements

- functionally-oriented complexity, 20
- object-oriented complexity metrics, 26

#### Metric Category

- functionally-oriented complexity, 20
- object-oriented complexity metrics, 26
- software builds, 42
- software defect reporting, 44
- software development, 33
- software effort, 49
- software problem resolution effort, 52

software schedule, 30

software testing, 38

#### Metrics Template and Field Descriptions

Figure 3-1, 12

#### Metrics/Phase Coverage

Figure 1.1-1, 5

MIL-STD-498, 11

MITRE Corporation, 3

module design complexity, 21, 25

### —N—

new code counts, 15

new requirement/enhancement, 68

#### Notes

- functionally-oriented complexity, 25
- object-oriented complexity metrics, 29
- software builds, 44
- software defect reporting, 48
- software development, 34
- software effort, 49
- software problem resolution effort, 52
- software schedule, 32
- software size, 17
- software testing, 41

number of children, 27, 28

### —O—

object, 28

cohesion, 29

coupling, 29

object-oriented complexity metrics, 26

oriented, 29

operations error, 67

organization, 4

organizational techniques, 8

originator, 69

over-expenditure of staff hours, 50

### —P—

percentage of public and protected data, 28

physical configuration audit, 4

polymorphism, 26, 29

post deployment software support, 3, 11

preliminary design review, 4

priority, 68

private, 29

#### problem

and defect reporting, 65

ID, 66

resolution effort, 3

status, 66

type, 67

undetermined, 68

problem status dates, 69

product ID, 66

progress, 1, 3

- development, 33
- software builds, 42
- testing, 38

projected availability, 69

public, 29

#### Purpose

- functionally-oriented complexity, 20
- object-oriented complexity metrics, 26
- software builds, 42
- software defect reporting, 44, 46
- software development, 33
- software effort, 49
- software problem resolution effort, 52
- software schedule, 30
- software size, 14
- software testing, 38

### —Q—

quality, 1, 2, 27, 28, 65

- problem resolution effort, 52

### —R—

references, 57

released / shipped, 69

reporting, 6

requirements defect, 67

resources, 1, 2

- size, 14

### —S—

schedule, 65

scope, 3

selection criteria, 4

size, 2

sizing templates, 14

SLOC. *See* source line of code

software

- build
  - indications, 44
- build progress - 3.6.2-1, 43
- builds, 3, 42
- class/build progress - Figure 3.6.2-2, 45
- complexity, 20
- defect questions, 48
- defect report guidelines, 65
- defect reporting, 44
- defect reports - Figure 3.7.2-1, 47
- defects, 2
- development, 33
- development drogress (Object-Oriented) - Figure 3.4.2-2, 36
- development progress - Figure 3.4.2-1, 35
- effort, 49
- measurement process concepts, 7
- measures, 11
- problem report, 46

- problem report attribute descriptions, 66
- problem report attributes, 65
- problem resolution effort, 52
- problem resolution effort questions, 53
- requirements review, 4
- schedule, 30
- schedule - Figure 3.3.2-1, 31
- size, 14
- size - Figure 3.1.2-1, 16
- test progress - Figure 3.5.2-1, 40
- testing, 38
- testing indications, 41
- testing questions, 41
- unit cyclomatic complexity, 25

Software Engineering Institute, 3

software staff, 49

source line of code, 8, 14

#### Special Definitions

- functionally-oriented complexity, 25
- object-oriented complexity metrics, 28
- software builds, 44
- software defect reporting, 46
- software development, 34
- software effort, 49
- software problem resolution effort, 52
- software schedule, 32
- software size, 17
- software testing, 41

special procurement requests, 6

SPR

- data formats, 48
- Density, 46

standard template, 11

strategic objectives, 1

system

- design review, 4
- requirements review, 4

### —T—

Table 1-1. The Metrics, 2

terms, 63

#### test

- case defect, 67
- effectiveness, 41
- efficiency, 39
- readiness review, 4

testing, 3

#### Tracking Period

- functionally-oriented complexity, 21
- object-oriented complexity metrics, 27
- software builds, 42
- software defect reporting, 46
- software development, 33
- software effort, 49
- software problem resolution effort, 52
- software schedule, 30
- software size, 14
- software testing, 39

## Triggering Guideline

- functionally-oriented complexity, 24
- object-oriented complexity metrics, 28
- software builds, 42
- software defect reporting, 46
- software development, 34
- software effort, 49
- software problem resolution effort, 52
- software schedule, 32
- software size, 17
- software testing, 41

## —U—

uniqueness, 68

unit development folders, 8

## Usage

- functionally-oriented complexity, 21
  - object-oriented complexity metrics, 27
  - software builds, 42
  - software defect reporting, 46
  - software development, 33
  - software effort, 49
  - software problem resolution effort, 52
  - software schedule, 30
  - software size, 14
  - software testing, 39
- user error, 67

## —V—

Valid SPRs By Phase Found and Category

Figure 3.9.2-3, 56

## —W—

Why Use Software Measurement, 7



THIS PAGE INTENTIONALLY LEFT BLANK